

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-2001

Using a Distributed Object-Oriented Database Management System in Support of a High-Speed Network Intrusion Detection System Data Repository

Phillip W. Polk

Follow this and additional works at: <https://scholar.afit.edu/etd>

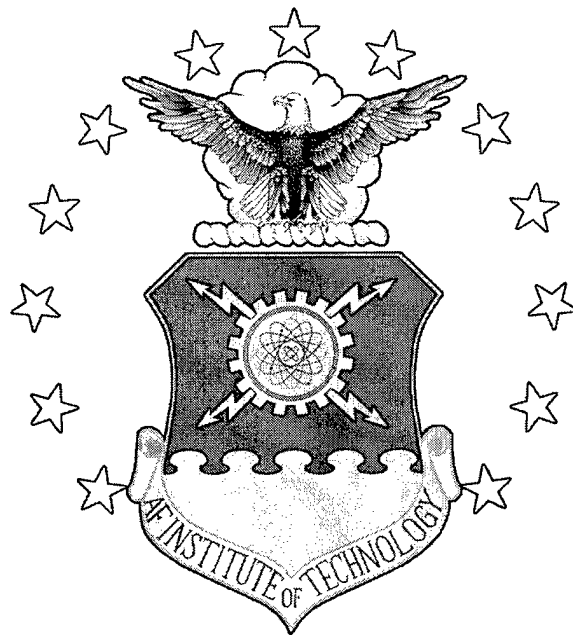


Part of the [Databases and Information Systems Commons](#)

Recommended Citation

Polk, Phillip W., "Using a Distributed Object-Oriented Database Management System in Support of a High-Speed Network Intrusion Detection System Data Repository" (2001). *Theses and Dissertations*. 4676.
<https://scholar.afit.edu/etd/4676>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.



USING A DISTRIBUTED OBJECT-ORIENTED
DATABASE MANAGEMENT SYSTEM IN SUPPORT
OF A HIGH-SPEED NETWORK INTRUSION
DETECTION SYSTEM DATA REPOSITORY

THESIS

Phillip W. Polk
2Lt, USAF

AFIT/GCS/ENG/01M-09

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

20010706 156

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

AFIT/GCS/ENG/01M-09

USING A DISTRIBUTED OBJECT-ORIENTED DATABASE
MANAGEMENT SYSTEM IN SUPPORT OF A HIGH-SPEED
NETWORK INTRUSION DETECTION SYSTEM DATA REPOSITORY

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Phillip W. Polk, B.S.

2Lt, USAF

March 2001

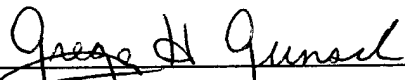
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCS/ENG/01M-09

USING A DISTRIBUTED OBJECT-ORIENTED DATABASE
MANAGEMENT SYSTEM IN SUPPORT OF A HIGH-SPEED
NETWORK INTRUSION DETECTION SYSTEM DATA REPOSITORY


Phillip W. Polk, B.S.
2Lt, USAF

Approved:



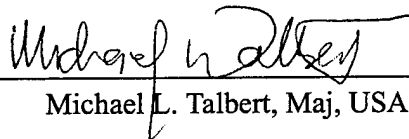
Gregg H. Gunsch (Chairman)

21 FEB 01
date



Karl S. Mathias, Maj, USAF (Member)

21 Feb 01
date



Michael L. Talbert, Maj, USAF (Member)

21 Feb 01
date

Acknowledgements

I would first like to thank my advisor, Dr. Gregg Gunsch, for his dedication and guidance throughout the lifecycle of this thesis. His time and effort were invaluable to the success of this research and my growth both as a student and as a junior Air Force officer. I would also like to thank my committee members, Major Michael Talbert and Major Karl Mathias for their database expertise and knowledge. Their lectures and advice led to a great understanding into the areas in which I probed.

I also would like to thank those at AFRL who allowed us to use their system as a model for the design portion of this thesis. Without Vinnie Salerno, the AIDE database and its functionality would have never been fully understood. Chet Maciag, Julia Pilny, Phil Zoleski, Greg Drew, Tom Daley, and Brian Spink also provided much needed information and support for this thesis. My thanks also to Dawn Vu from AFIWC/CMET. I hope that AFIT students can continue to work with individuals such as these.

My thanks goes out to Abraham Glazer, Jack Murray and Robert Cheong from Objectivity, Inc. for providing the resources necessary to deploy their database products and keep them running.

And to my wife and children, who have endured the long nights of study and research, I dedicate this thesis. Without their patience and understanding, none of this would have been possible.

Table of Contents

	Page
Acknowledgements	iv
List of Figures	viii
List of Tables	x
List of Abbreviations and Symbols	xi
Abstract	xiii
CHAPTER I. Introduction	1
1.1 Background	1
1.1.1. ASIM/CIDDS.	2
1.1.2. AIDE.	2
1.2 Problem Statement.	3
1.3 Research Hypothesis	6
1.4 Scope	9
1.5 Assumptions.	10
1.6 Thesis Organization	10
CHAPTER II. Literature Review.	12
2.1 Current USAF Supported IDS Data Repository Initiatives: ASIM/CIDDS & AIDE	12
2.1.1. ASIM.	13
2.1.2. CIDDS.	15
2.1.3. AIDE.	16
2.1.4. System Comparison	22
2.2 Object Persistence	23
2.2.1. What is an object?	24
2.2.2. Relational vs. Object-Oriented Implementations.	29
2.3 RDBMS to OODBMS Schema Transformation	38
2.4 Federated OODBMS	39
2.4.1. Distributed DBMS (DDBMS)	40
2.4.2. Distributed Object-Oriented DBMS (DOODBMS) Design	43
2.4.3. Tightly-coupled vs Loosely-coupled DDBMS	44
2.4.4. Federated DBMS	45
2.4.5. DOODBMS Design	45
2.5 Objectivity/DB Overview	50
2.5.1. Architecture.	50
2.5.2. Development	52
2.5.3. Transactions	55

2.5.4. Security	56
2.5.5. Distribution	56
2.5.6. Schema Evolution	57
CHAPTER III. Methodology	58
3.1 Introduction	58
3.2 DOODBMS Design	58
3.3 Database Integration into IDS Application	64
3.3.1. Components	65
3.4 Summary	67
CHAPTER IV. Database/System Implementation	68
4.1 Introduction	68
4.2 Analyze IDS Structure	68
4.3 Re-engineering the RDBMS Schema to a DOODBMS Schema	70
4.3.1. ER Model of AIDE	71
4.3.2. AIDE Initial Object Model	71
4.3.3. Refine Classes	71
4.3.4. Prepare a Functional Model	74
4.4 High-Level Design	75
4.4.1. System Design Trade-offs	76
4.4.2. Resolve to-Many Relationships with Respect to Database Representation	76
4.4.3. Refine Classes and Associations to Better Model Data	77
4.4.4. Design Global Conceptual Schema	78
4.4.5. Distribution Design and Local Schema Design	79
4.4.6. Implementing Using Objectivity/DB	82
4.5 Low-Level Design	89
4.6 Tap Application Development	91
4.6.1. Conceptualization	91
4.6.2. Analysis	91
4.6.3. System Design with OODBMS and Detailed Design	95
4.7 Summary	96
CHAPTER V. Testing and Evaluation	97
5.1 Performance Benchmark	97
5.1.1. Testing Steps and Architectures	98
5.1.2. OOAIDE Real Secure Test Results	103
5.2 Database Distribution	110
5.3 Demonstrate Taps Utilizing Common OOP Design	111
5.4 Summary	112
CHAPTER VI. Conclusions	113
6.1 Implementation Critique	113
6.1.1. AIDE to OOAIDE Translation Limitations	113
6.1.2. Accomplishments	114

6.1.3. Disadvantages	115
6.2 Future Research	116
6.3 Conclusion	118
Appendix A. AIDE Data Dictionary	119
Appendix B. OOAIDE Data Dictionary	125
Appendix C. Real Secure Tap/Trigger Code	129
C.1 Perl/Oracle AIDE Real Secure Tap Code	129
C.2 Java OOAIDE Real Secure Tap Code	130
Appendix D. Query Test Code	133
D.1 All Events	133
D.1.1 AIDE	133
D.1.2 OOAIDE	134
D.2 All Events with AIDE Signature	135
D.2.1 AIDE	135
D.2.2 OOAIDE	137
D.3 Events of Single IP	138
D.3.1 AIDE	138
D.3.2 OOAIDE	139
D.4 Events of Single IP with Sensor	140
D.4.1 AIDE	140
D.4.2 OOAIDE	142
Bibliography	143
Vita	146

List of Figures

Figure	Page
2-1. ASIM/CIDDS High-Level Architecture	13
2-2. AIDE Tap/Bridge Architecture Diagram	19
2-3. Layout of Sites and Hosts	20
2-4. AIDE ER Diagram	21
2-5. Generalization Example	28
2-6. Object to Relational Mapping Using OID	30
2-7. Memory Hierarchy usingOODBMS	34
2-8. Objectivity/DB OID Composition	51
4-1. OO-AIDE Process Diagram	69
4-2. AIDE ER Diagram	70
4-3. AIDE Entity-to-Class Translation	72
4-4. Refined Class Diagram	73
4-5. Final Class Diagram	75
4-6. Global Schema	78
4-7. Quorum and Non-Quorum with Unevenly Distributed Weights	83
4-8. OOAIDE Databases and Partitions	85
4-9. OOAIDE Database Containers	87
4-10. OOAIDE Class Diagram	95
5-1. Real Secure Tap Architecture	98
5-2. Object Instantiation and Record Insertions for AIDE and OOAIDE Tap	104

Figure	Page
5-3. Queries on AIDE and OOAIDE via Java	105

List of Tables

Table	Page
2-1. Graphical Representation of Class Using UML.....	26
2-2. RDBMS vs. OODBMS Comparison	29
2-3. Extensions Needed When Utilizing an RDBMS	32
2-4. A C I D	42
5-1. Real Secure Database Fields	103
5-2. Hardware/Software Test Configuration	104
6-1. Perl Specifications for Real Secure Tap.....	129

List of Abbreviations and Symbols

AFED	Air Force Enterprise Defense
AFIWC	Air Force Information Warfare Center
AIDE	Automated Intrusion Detection Environment
AMS	Advanced Multi-threaded Server (Objectivity/DB)
ASIM	Automated Security Incident Measurement
CIDDS	Common Intrusion Detection Director System
CMET	Countermeasure Engineering Team
DDBMS	Distributed Database Management System
DDL	Data Definition Language
DOODBMS	Distributed Object-Oriented Database Management System
DRO	Data Replication Option (Objectivity/DB)
EPIC	Extensible Prototype for Intrusion Control
ER	Entity-Relationship
FDBMS	Federated Database Management System
FTO	Fault Tolerance Option (Objectivity/DB)
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IDT	Intrusion Detection Tool
IP	Internet Protocol

JDBC	Java Database Connectivity
LS	Lock Server (Objectivity/DB)
MROW	Multiple Readers, One Writer
NFS	Network File System
NIDS	Network Intrusion Detection System
ODBC	Open Database Connectivity
ODMG	Object Database Management Group
OID	Object Identifier
OMT	Object Modeling Technique
OO	Object-Oriented
OOA	Object-Oriented Analysis
OOAIDE	Object-Oriented Automated Intrusion Detection Environment
OOD	Object-Oriented Design
OODBMS	Object-Oriented Database Management System
OOP	Object-Oriented Programming
RDBMS	Relational Database Management System
SLAC	Stanford Linear Accelerator Center
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
VPN	Virtual Private Network
XML	eXtensible Markup Language

Abstract

The Air Force has multiple initiatives to develop data repositories for high-speed network intrusion detection systems (IDS). All of the developed systems utilize a relational database management system (RDBMS) as the primary data storage mechanism. The purpose of this thesis is to replace the RDBMS in one such system developed by AFRL, the Automated Intrusion Detection Environment (AIDE), with a distributed object-oriented database management system (DOODBMS) and observe a number of areas: its performance against the RDBMS in terms of IDS event insertion and retrieval, the distributed aspects of the new system, and the resulting object-oriented architecture.

The resulting system, the Object-Oriented Automated Intrusion Detection Environment (OOAIDE), is designed, built, and tested using the DOODBMS Objectivity/DB. Initial tests indicate that the new system is remarkably faster than the original system in terms of event insertion. Object retrievals are also faster when more than one association is used in the query. The database is then replicated and distributed across a simple heterogeneous network with preliminary tests indicating no loss of performance. A standardized object model is also presented that can accommodate any IDS data repository built around a DOODBMS architecture.

USING A DISTRIBUTED OBJECT-ORIENTED DATABASE
MANAGEMENT SYSTEM IN SUPPORT OF A HIGH-SPEED
NETWORK INTRUSION DETECTION SYSTEM
DATA REPOSITORY

I. Introduction

Network Intrusion Detection Systems (NIDS) are tools used to monitor traffic to and from a network in order to provide information on any attempts by intruders to gain access to a network system. Currently, there are two major NIDS initiatives within the United States Air Force (USAF). One is the Automated Security Incident Measurement (ASIM) and Common Intrusion Detection Director System (CIDDS) developed by the Countermeasure Engineering Team (CMET) under the Air Force Information Warfare Center (AFIWC). Both ASIM and CIDDS are deployed in support of the AFIWC to provide network intrusion reporting and advanced analysis of collected network data from across the Air Force. The second initiative is the Automated Intrusion Detection Environment (AIDE) developed by the Air Force Research Laboratory (AFRL) in support of the DoD.

1.1 Background

Both ASIM/CIDDS and AIDE are similar in the way they process data. Packets are retrieved from the physical network through either a proprietary network sniffing method, which is the case with ASIM, or commercial IDSs and packet sniffers, as with AIDE.

The data is then converted into some usable form for the system so near-real-time alerts may be generated on known attempt signatures. ASIM produces C++ and Java objects, and AIDE feeds the IDS data into scripts or C programs before sending it to the database. The ASIM objects, and resulting AIDE data, are then inserted into their respective databases.

Analysis and reporting tools are used to dissect the data in the databases and form an overall picture of current network status. Both systems utilize Java interfaces as a means to display information from the database to the analyst. Using tables and in AIDE's case, maps and bar charts, the Java front-end also allows the analyst to manipulate and insert certain pieces of information.

1.1.1. ASIM/CIDDS.

ASIM/CIDDS went into development in 1993 as a beta project funded by the USAF to test the plausibility of deploying such a system on its networks. Since then it has become a full-featured NIDS and functions in a production environment.

Using a "black-box" approach, a stand-alone sensor unit is deployed to a remote site and connected to the network between the Internet and the USAF base's network where it passively sniffs packets as they transition from the public to private network. Data is then sent from the sensor to a director under which it is assigned. Once at the director level, the data is converted and inserted into CIDDS, the database that serves as the analysis and reporting component of the current USAF IDS and which continues to revolve around the Oracle Database Management System (DBMS). A Java console is used to extract and manipulate information reported to an analyst [Veridian 9-10].

1.1.2. AIDE.

AIDE, as opposed to CIDDS, remains a research project developed under AFRL in Rome, NY and is not currently used as a production system. However, it has been deployed

to approximately twelve DoD locations for testing purposes. The system itself operates much like that of ASIM/CIDDS, but does not incorporate a proprietary means of pulling data off of the physical network. Instead, it integrates output from a number of commercial IDSs to populate its Oracle database, which makes up the bulk of the system. For this reason, AIDE can be considered an *IDS data repository* vice a true NIDS, as is the case with ASIM/CIDDS. Once the packet data is in the database, analysis can provide a complete picture of network intrusions regardless of the information's source. Detailed reports and visual models are employed to better represent the information.

1.2 Problem Statement

Since both systems are inherently focused around the database they employ, it would be a natural assumption that the database would be one of the potential bottlenecks within the overall architecture both in terms of performance and representation of data. Due to Oracle's relational structure, there are several areas that may affect the database's ability to sustain a level of performance consistent with the high-speed networks on which it is deployed:

- *Relational Management Overhead.* In both cases, a relational database is used and when data is inserted, table look-ups must be performed and foreign keys defined to maintain any 1:1, 1:N, or N:M relationships inherent to the information. This means that the ASIM objects and AIDE data to be inserted must be broken down and additional identifiers must be produced to uniquely identify each set of data.
- *Persistence Storage vs. Transaction Performance.* Data persistence refers to the idea of accessing information even after the program that created it has terminated. Databases, in general, provide a storage mechanism to achieve persistence [Loomis 17]. As previously indicated, one of the main problems with relational data-

bases involves the overhead in storing and retrieving information to achieve data persistence when table look-ups and joins are performed. Both the ASIM and AIDE systems must convert data from the physical network into a form that can be inserted into a relational database for storage. In ASIM, the conversion process takes the data off the wire and builds an object from the data using C++. The next conversion occurs when the object is transmitted from ASIM to CIDDS via a Java communication program. Finally, the object is broken down and inserted into the database, in which case the data within must be mapped to the relational database structure through table lookups and insertion of foreign keys. Foreign keys are used by a relational database designer to link tables together to form relationships and may either be created by the designer/programmer or automatically by the DBMS. This object-relational mapping is relatively slow when compared to other storage techniques [Larson 209]. In terms of performance, the database in the AIDE system can handle 150-200 transactions/sec. In ASIM, packet data are inserted into the database at several thousands of transactions/sec.

- *Object-Oriented (OO) Programming Integration.* RDBMSs revolve around tables. Using the Structured Query Language (SQL), a program may insert and query the information contained within the RDBMS tables. This means that if a programmer using an object-oriented programming language is to interface with the database to manipulate data, SQL must be embedded somewhere within the code to translate an object's attributes to the table structures defined within the

RDBMS. The SQL statements themselves can require a substantial amount of code, to include the code needed to convert data types used within the object to data types that the RDBMS can store.

- *Complex Data Types.* Relational databases are only able to store data of a predetermined type. When a database ships for market, the data types (e.g., int, char, string, memo, etc.) are embedded into the system and any data inserted into the database's tables must be broken down to match these types. In the application domain, types may include not only those that the database can work with, but also more complex data types developed by the programmer to handle many different kinds of information across various domains. One example is the use of inheritance. An *Officer* object may be of type *Person*, in which case the programmer would most likely save the *Officer* attributes in the database under the table OFFICER. However, the semantic information, namely that which shows the hierarchical inheritance structure, has disappeared and it is unclear when looking at the database whether or not an *Officer* inherits attributes from *Person* or is completely independent [Loomis 77]. The logic code for determining this inheritance hierarchy must be inserted into the code so that when an *Officer* is read back into the application from the database, it is instantiated as type *Person*. This also contributes to the increase in code size associated with embedded SQL statements.
- *Distributed Systems.* A distributed database has yet to be deployed in either system and will undoubtedly cause additional problems with administration. It is unclear how Oracle will handle the load when working across multiple sensors on

a distributed network considering the basic difficulties of partitioning, data replication, and schema evolution [Larson 57-58]. Distribution will allow the data to be replicated to deal with *single points of failure*. As it stands, data is centrally stored at a particular site. However, to insure data availability, data needs to be replicated between multiple sites within the hierarchy so that if one director's data is unavailable, it may still be retrieved from a mirror site on the network.

1.3 Research Hypothesis

In dealing with the above problems and needs, a new storage technique is desired which can increase performance, allow the system to be easily distributed, and eliminate many of the data conversions necessary when dealing with a relational database. One alternative would be an object-oriented database management system (OODBMS). My hypothesis is that a distributed OODBMS (DOODBMS) will provide the following enhancements over the current and conventional RDBMS:

- Improved performance in inserting events, and subsequently, other data into the database. The research in this thesis primarily involves insertion throughput. However, query throughput is also tested.
- Allow for distribution of the overall system to alleviate single points of failure resulting from a loss of database connectivity.
- Provide a standardized OO programming base for the system as a whole. By standardizing to the OO model, developers will not be faced with maintaining programs written in multiple languages without a model guiding the process. In the AIDE system, for example, programs to retrieve data are written in Perl, Java, C,

Oracle ProC, lex/yacc, and others. The use of a DOODBMS and common OO structure that must exist to utilize it will allow for more software reuse and easier maintenance of the entire system.

The following are ways in which a DOODBMS may be used to address specifically the limitations and problems with the RDBMS:

- *Data Persistence.* DOODBMS' primary goal is to provide object persistence.

When a program is executed, instantiated objects may be either transient or persistent. In the case of persistent objects, objects are saved to permanent storage so as to survive the termination of the program in which they were instantiated. When using a DOODBMS in an object-oriented programming (OOP) language, the programmer may deem an object persistent and from that point on does not care whether or not the object is in memory or being retrieved from disk. In essence, data on disk mimics that in memory and no additional mapping is needed to form associations between data objects since the entire hierarchy is being maintained. This is known as single-level memory [Kroenke 490-491]. In relating this to ASIM/CIDDS and AIDE, it is possible to build a sensor that can take packets off of the physical network and convert them into persistent objects almost immediately. From that point on, provided that the objects were successfully inserted into the DOODBMS, the objects are available to other processes regardless of the status of the process that placed them there. As a result, there is far less overhead associated with placing objects into the database than there is in mapping them to a particular relational storage schema through lookups and foreign key generation.

- *OO Program Integration.* The DOODBMS also provides programming interfaces that are fully integrated with OOP languages. In the case of Objectivity/DB, the DOODBMS used in the programming portion of this thesis, interfaces are provided for C++, Java, and Smalltalk. As a result, the programmer can build relationships within the context of the language used, such as using arrays of pointers and references, and need not convert to 1:1, 1:N, or N:M relationships consisting of foreign keys and other constructs to maintain the structure of the complex data. However, it is understood that SQL queries may be necessary in some programming environments and non-object-oriented applications may need access to data. For this reason, a SQL parser and ODBC package is included with Objectivity/DB.
- *Distribution through a Federated Database.* Objectivity/DB also provides for a distributed architecture known as a federated database. In such a system, the database is a loosely-coupled distributed DBMS, but with certain facilities not implemented to allow for easy administration from a single node and local processing of data [Larson 46]. This means that schema changes can be made available at one site and propagated down through the hierarchy while older data, conforming to the older schema, are still maintained through a process known as *versioning*. Databases may also be coupled from site to site so that objects manipulated at one site are updated at all of the others. This is known as *data replication*, which aids in *fault tolerance*. In an IDS, the data that comes in from the sensors is managed and stored locally at that sensor's director. When distributed, directors may have

full access to any other director's database located within the federation and might also replicate the data amongst multiple databases, thus avoiding a single point of failure.

1.4 Scope

This research will consist of replacing the Oracle database used in AIDE with Objectivity/DB and analyzing the overall performance characteristics within a controlled environment. To effectively measure the performance increase and the effect of distributing the database across a heterogeneous network, the following activities must be demonstrated:

- Sensor monitoring and data retrieval from logs and other data stores kept by a variety of sensors
- Passing of information to both a single site and multiple distributed sites
- Insertion of data into the database with both the data replication and fault tolerance features implemented
- Retrieval of information via Java tools

As an additional contribution to the system, the object model will establish an OO base to be used by the database and applications accessing the data. The OO base allows designers to provide a solid architecture to programmers and database administrators, which aids in both maintenance and software evolution.

Both ASIM/CIDDS and AIDE are systems that may benefit from DOODBMS technology. In fact, this approach may be applicable to any IDS data repository that uses OO programming and may need to deploy a DOODBMS as a robust, high-speed database solu-

tion. Due to the availability of data, database schema, scripts, and other crucial pieces of information by AFRL, modeling of the OO system using a DOODBMS will revolve around requirements for future AIDE releases.

1.5 Assumptions

- Standardized test data will be produced locally by ISS's Real Secure NIDS analyzing streams of test packets produced by Nessus. [RealSecure; Nessus]
- The object model and DOODBMS architecture will be built with the assumption that the data will eventually be distributed across a number of systems monitoring multiple sensors.
- Agents, XML, and other object-related systems will be used in the future to report and use information stored in the database. C++, XML, and Java are the program languages and structures of choice at this point. Although this seems to go against the hypothesis that using multiple languages is undesirable, what is of major importance is that the object model and objects instantiated by the system remain intact regardless of whether or not they are currently used by an analyst or stored in the database.

1.6 Thesis Organization

The thesis is divided into six chapters. Chapter I states the hypothesis and gives background into the problems facing the IDS data repositories currently being used. Chapter II gives additional background information and provides knowledge necessary for understanding the various systems, databases, and architectures used throughout this thesis. Chapter III describes the methodology used to rebuild the AIDE system into one that uses the DOODBMS. Chapter IV demonstrates the used of the methodology by developing the

Object-Oriented Automated Intrusion Detection Environment (OOAIDE). Chapter V tests OOAIDE by inserting and retrieving information, and presents observations of the OO application and the system's distributed architecture. Finally, Chapter VI presents the conclusion giving the results of this research and additional future areas of research involving OOAIDE.

II. Literature Review

This chapter describes the two USAF IDS data repositories that serve as candidates for conversion to DOODBMSs, reviews literature describing and contrasting the functionality of RDBMSs and OODBMSs, provides documentation on basic OO design and distributed database features important to an IDS, and gives a general overview of Objectivity/DB.

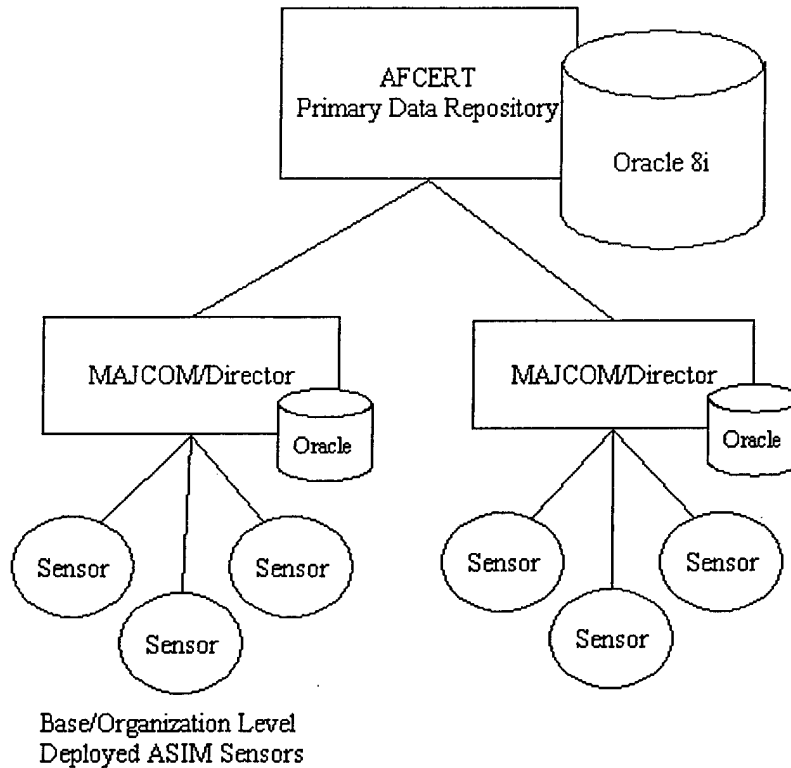
2.1 Current USAF Supported IDS Data Repository Initiatives: ASIM/CIDDS & AIDE

This thesis will concentrate primarily on two of the USAF conducted research initiatives involving intrusion detection. Although the two initiatives are very similar in nature, a few differences are discussed that set each apart from the other. These two projects were chosen over COTS products because of their ability to provide unique solutions to two current USAF data collection needs. ASIM/CIDDS provides the ability to collect large amounts of data concerning every connection made to any government system on the networks that it protects for forensics purposes and AIDE has the ability to consolidate intrusion related data from a variety of sensors already deployed within the DoD on heterogeneous networks.

The Air Force's primary IDS/IDS data repository, ASIM/CIDDS, consists of two distinct subsystems that are completely dependent on one another. Both subsystems consist of dedicated hardware and software packages that are used to support the AFIWC Intrusion Detection Tools (IDT) program. The architecture of the overall IDS is designed around a hierarchical tree-like structure with a node at the Air Force Computer Emergency Response Team (AFCERT), which acts as the primary data store and fusion center for all network information collected by the various MAJCOMs. Under each MAJCOM lies a number of ASIM sensors that remotely collect and send information, and possible alerts, back for fur-

ther analysis. The descriptions presented here are inherent to version 3.0 of both ASIM and CIDDs, which were being deployed at the time of this writing.

Figure 2-1. ASIM/CIDDs High-Level Architecture



2.1.1. ASIM

The USAF standard for the IDS, ASIM, acts as a remote sensor to CIDDs. The sensor sits at the front of a network and is the first network asset to intercept a stream of data before it is routed to its next destination. As a sniffer, it passively pulls data from the wire. As an IDS, it then analyzes the data to look for signatures defined within its rule set that constitute a violation of network policy. The sensor operates in real time and alerts of an intrusion attempt are passed to CIDDs.

ASIM is associated with two different representations of the same data. The raw data, which are packets kept in non-human readable form, are stored locally to the ASIM hard drive and optical removable media. CIDDs is required if an operator or analyst needs

to remotely retrieve and convert the raw data to ASCII. The second type, connection data, is parsed from the raw and sent to CIDDS located at one of twelve MAJCOMS on a Virtual Private Network (VPN) connection via the Java Communications Process (JCP). The connection data contains information concerning the actual interaction of one computer to another, such as source and destination Internet Protocol (IP) addresses, source and destination port numbers, protocol used throughout the transaction, and the duration and status of an existing connection. Load balancing and fault tolerance features are built into ASIM to detect when a director is unavailable for data retrieval due to an outage or saturation, in which case another director is used as its storage medium.

ASIM hardware architecture centers on the SecureCom 6000 module SC6000-DUAL. According to Intrusion.com,

“The SecureCom 6000 family is a flexible chassis-based system that integrates network and application servers, firewall, VPN, intrusion detection, routing, and LAN ports into a single compact, fault-tolerant platform. The platform can be configured to provide whatever functionality your LAN requires in a manageable and coherent structure.” [SecureCom]

In other words, the system functions as a black-box solution. The SC6000 module used in ASIM version 3.0 contains:

- Dual 600MHz Pentium III Processors
- 256Mb RAM
- FreeBSD Operating System
- 18Gb Hard Drive
- 100 Base-T Ethernet Connectivity

2.1.2. CIDDS

CIDDS is a much larger system consisting of an Oracle 8i database, Java Graphical User Interface (GUI), and some degree of Artificial Intelligence (AI). Its main function is to retrieve information sent to it by a number of ASIM sensors located throughout the operational Air Force on independent homogenous or heterogeneous networks. The subsystem simply gives ASIM a centralized, long-term storage capability and allows USAF analysts to correlate data to detect network intrusion attempts, possible misuse violations, and other unauthorized activities. Furthermore, it functions as a trend analysis and historical database for activities that involve multiple networks or that constitute attacks over long periods of time.

CIDDS is concerned with connection and event-related data routed from ASIM. Upon receiving connection data, CIDDS either parses the stream so that it may be inserted into the Oracle 8i database using standard SQL or, when the data is more critical, forwards the data to a higher-level director, such as AFCERT. Any incoming alerts are routed to the GUI for analysts to view. CIDDS is, in all practicality, the workbench and storage mechanism for the analyst. The database stores information on protected domains, sensor locations, connections between systems passing through USAF networks, actual events or alerts, and histories of events for trend analysis.

2.1.2.1. CIDDS Database

The Oracle 8i database used to maintain data for ASIM and CIDDS is of primary concern in this thesis. Without the database, the system is relatively useless when analyzing and correlating events.

There were many optimizations made over the basic installation of Oracle 8i to get the CIDDS database to the point of handling several thousand transactions per second. First, the database is partitioned day-to-day to limit the search space that a query will make in finding data. For example, if an analyst wishes to see data collected 48 hours previously,

then the database would map the query to the partition on the disk that contained the data versus searching the entire database. In effect, the partitioned disk becomes a circular queue. An index is stored on the same partition that contains the data that it indexes, thus further limiting the search space.

PL/SQL, Oracle 8i's procedural language that enables a database administrator to mix SQL with program units such as procedures, functions, and packages, is also used to optimize some of the queries that take place on the database [Oracle Glossary-6]. Stored procedures are used in conjunction with the database cursor to keep join overhead to a minimum during information retrieval by keeping certain predetermined pieces of information cached in memory so they do not need to be retrieved from disk when a query is issued from the GUI Server. The same procedures cache and store alert data from ASIM even before it is inserted into the database so that an alert may be delivered to an analyst's GUI without the analyst having to access the database tables to retrieve the information. This is an important optimization, since disk retrievals are more expensive in terms of time when compared to memory access.

2.1.3. AIDE

The second distributed IDS initiative, AIDE, is being developed as the next generation system for the Department of Defense (DoD) under AFRL. As part of the Extensible Prototype for Intrusion Control (EPIC) program at AFRL, which includes such systems as the Air Force Enterprise Defense (AFED) initiative, AIDE serves as a in-depth detection system in "identifying deviations from normal operational states in the enterprise in real time and predictively from network, computer, and open-source indicators [AIDE]." Although similar to the ASIM/CIDDS architecture, there are a number of characteristics that set this system apart from ASIM/CIDDS and other COTS solutions.

Unlike ASIM/CIDDS, the primary focus of AIDE is not the interception of network traffic for analysis. AIDE's main goal is to fuse data from multiple sources to build a com-

plete picture of current network status and intrusion attempts. The raw data itself has already been captured, parsed, and processed by one or more commercial IDSs or network tools and just needs to be imported into the database for storage, correlation, and analysis. Such network tools may include routers, IDSs, firewalls, or any other mechanism capable of producing local log entries, streams of data, or database entries based on sensor reported information. CIDDs itself is one source of information for AIDE.

2.1.3.1. Architecture

The architecture of AIDE is relatively simple considering the task. When a network tool or IDS captures an event, a signature is produced by that system. There are a variety of methods by which the information ultimately gets entered into the database.

- *Taps and bridges.* A *tap* runs on the sensor machine that captures traffic and creates a connection to a *bridge* located on the Oracle database machine. The taps are small C, Perl, or lex/yacc programs that are built to use as few resources as possible so as not to interfere with a sensor's normal operation. The bridges are responsible for taking data, or signatures, read from the tap and parsing them for insertion into the database. A bridge and tap combination exists for most of the GOTS and COTS sensors or tools expected on the network, which may include Raptor, JIDS, Snort, and others [Raptor; JIDS; Snort]. Most of the data that is sent up via a tap resides in a file on the sensor system, which is read periodically by the tap.
- *Data Streams.* In some cases, data is streamed directly into a port at a site where the database is placed. Special bridges listen to those ports and intercept any incoming binary streams of information. Most often, routers and some firewalls,

such as Sidewinder, take advantage of this more direct route [Sidewinder]. However, some IDSs, such as NetRadar, also have the capability to stream data [NetRadar].

- *ODBC*. And the last method by which data is entered involves a simple connection to Oracle through ODBC. The Real Secure tap, which reads its data from an Access database, uses this method. In reality, the data may be extracted from both log files and other databases (Access, Oracle, SQL Server), parsed, and then inserted directly into Oracle.

The database then takes the data and uses a complex set of database triggers to filter it, determine which set of tables are involved, and complete any additional processing. Figure 2-2 shows an example of AIDE's architecture using the tap/bridge combination.

As seen in the diagram, once the data is in the database, the *correlator* is used to correlate data from multiple sensors so that the actual activity that produced the signatures can be deduced. For instance, a portscan may be conducted on a network from the outside. Several sensors may pick up this activity and enter it into the database as separate events. Instead of reporting each of these as separate attacks, the correlator correlates the findings of the sensors and hopefully reports them as a single portscan.

The actual layout of the systems is hierarchical in nature, much like that of ASIM/CIDDS. The databases are located at *sites* distributed throughout the DoD, which are normally associated with a base or organization. Below those sites lie *hosts*. A host is any machine, whether computer or router, that is located within the domain of the site. A *sensor* resides on a host. The database is set up to allow a host to run many sensors, but ordinarily only one sensor resides on a given host. Figure 2-3 gives a representation of a site and its hosts located on a typical network. In practice, all the hosts actually have information stored in the database. Through the use of a tool such as *nmap*, information can be gathered

from all the hosts on the network [Nmap]. That information may then be placed in the database to give a detailed look at all the systems in the domain, regardless of whether or not they will report any event activity.

Figure 2-2. AIDE Tap/Bridge Architecture Diagram

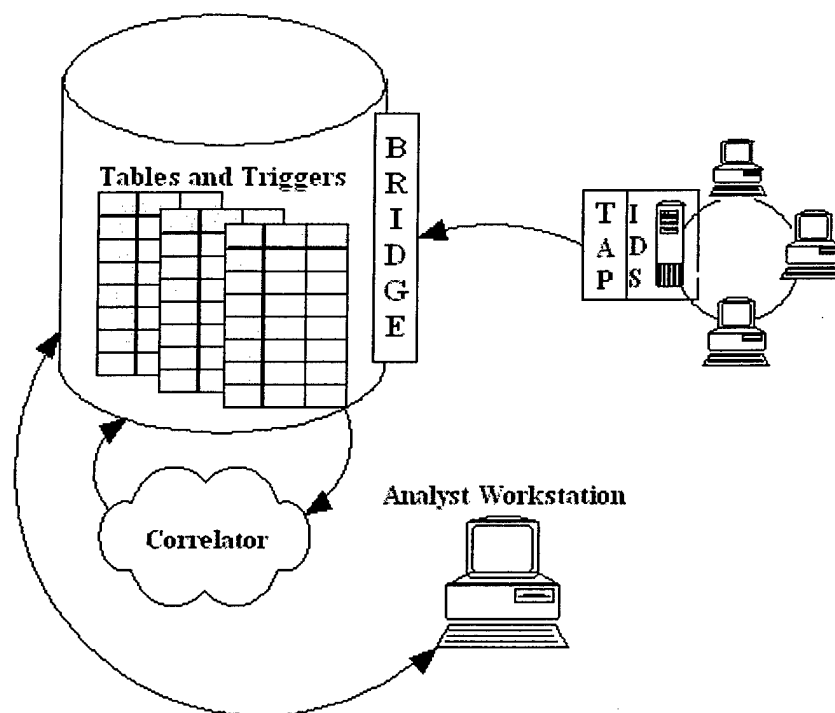
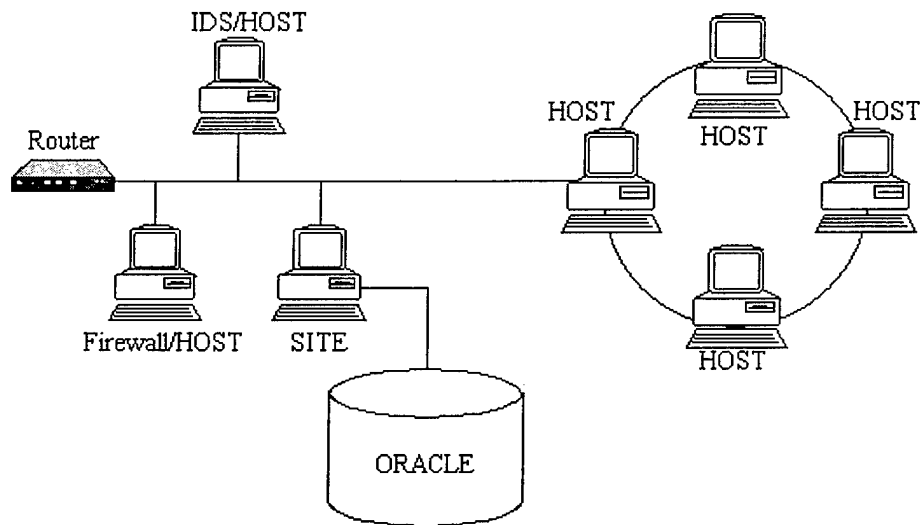


Figure 2-3. Layout of Sites and Hosts

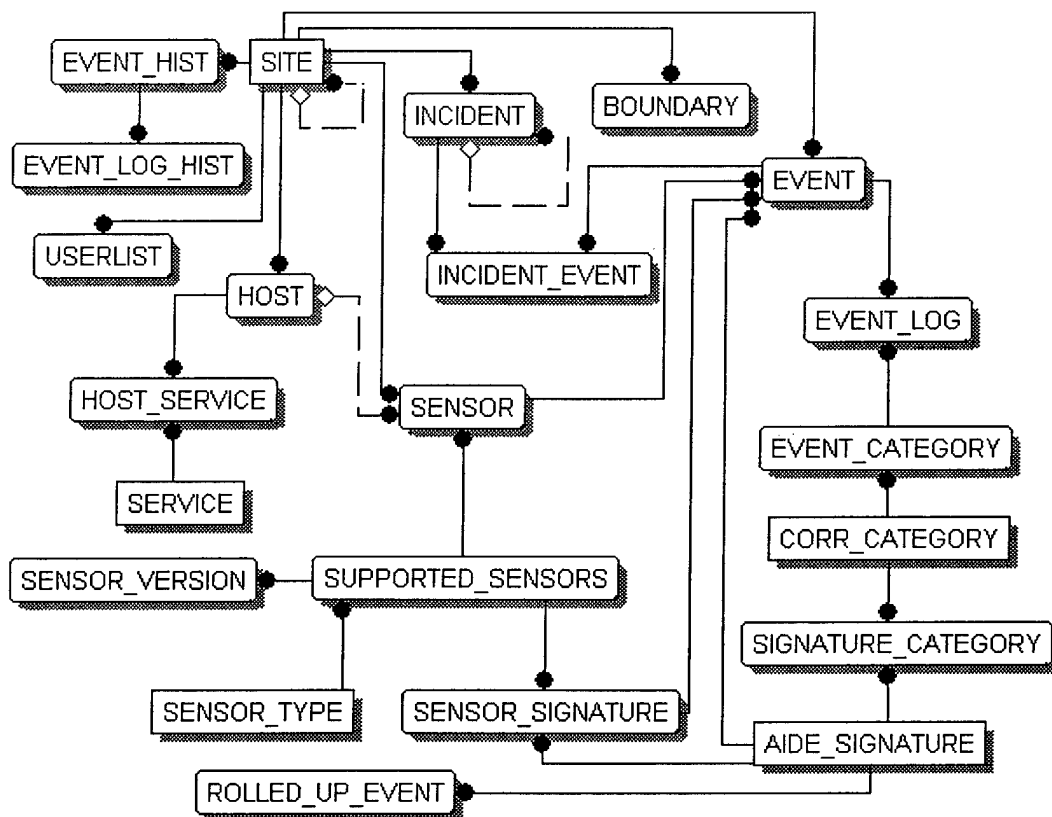


2.1.3.2. Database

The database currently being used with AIDE is also Oracle 8i. In keeping with AIDE's primary function, the smallest details concerning each and every connection are not handled by the system. Instead, AIDE concerns itself with keeping track of signatures and events thrown by other systems and correlating those signatures into a single, normalized AIDE signature.

There are a few enhancements made to the basic database installation, but not quite as many as in CIDDS. This is mainly because AIDE functions as a research system versus a full production system. However, there are strains on the database that are a result of many different sensors throwing multiple signatures simultaneously. To handle the input, AIDE uses a disk separated into seven separate partitions, one for every day of the week. As each partition is reused, all of its data is copied to a separate table located on a non-partitioned disk, which serves as long-term storage. This technique, as stated before when describing CIDDS, reduces the search space significantly. The tables used within the database do not differ significantly from those of CIDDS. Below, Figure 2-4 shows a simplified relational schema of the AIDE database.

Figure 2-4. AIDE ER Diagram



The database has two significant portions, SITES and EVENTS. All the other tables either establish a hierarchy to the information, such as a SENSOR belongs to a HOST, which belongs to a SITE, or tables that associate tables with each other. Association tables include INCIDENT_EVENT, EVENT_CATEGORY, HOST_SERVICE, and others that follow the same naming scheme. The unique portion of this schema, and the portion that separates it from that of CIDDS, is the portion involving the SENSOR_SIGNATURE and AIDE_SIGNATURE tables. Using these, along with the SIGNATURE_CATEGORY, multiple signatures from various IDS packages may be combined into a single AIDE signature to aid in data reduction. An example may be a number of IDSs that see a portscan on a network, but each has a different event code in their log file for reporting the event. Using the signature tables, AIDE combines those codes into a single signature and category

so that analysts do not concern themselves with the low-level information produced by the individual sensors.

To facilitate the need for information from a site other than the one to which an analyst initially connects to, *links* are used throughout the database that 'point' to tables in the databases located at other sites. In essence, data is only entered into the database once and it resides at that particular location. In doing this, the AIDE system may ensure the integrity of the data. This means that if a site goes down for any reason, then its data is also unavailable.

2.1.4. System Comparison

Both systems closely represent the common IDS architecture as proposed by Ranum [Ranum]. There are information-gathering mechanisms, storage for long-term or short-term analysis, and reporting engines in both implementations. As far as the database is concerned, there are even a number of similarities that allow one to see a common set of attributes being collected and stored. For instance, both keep information on hosts containing the systems and events thrown by those sensors. The particular fields collected contain IP addresses, types of attacks, and ports hit with the attack. There are only so many ways to parse packet information. However, there are major differences, some of which were pointed out earlier within the system descriptions, which make each system stand out in a particular environment.

ASIM/CIDDS are built for a high-speed environment where all possible connection data must be collected for forensics purposes. This has led to a very robust and optimized solution, in which both parts of the system are pushing the limits of both the hardware and software on which they run.

AIDE, on the other hand, is primarily suited for correlation of events where the network data has already been sifted through and, compared to the amount of traffic on the network, fewer entries need to be inserted into the database. This has allowed AIDE to sur-

vive without quite the number of optimizations that have been built into ASIM/CIDDS. It was mentioned earlier that ASIM/CIDDS has an estimated data storage rate of several thousand transactions per second. By contrast, AIDE tops out at about 200 entries per second due to its intense use of Oracle triggers for event management and correlation.

ASIM/CIDDS are considered a "closed" design. This simply means that they work hand in hand and with no other sensors or data collected. This will most likely change in the next release of the product, but it is how the system remains at the time of this writing. Everything input into the database is a result of the ASIM sensor. AIDE is designed to work with a number of systems, to include CIDDS, NetSquared's NetRadar sensor built under AF contract, DISA's JIDS, Symantec's Raptor, Secure Computing's Sidewinder, and ISS's RealSecure [JIDS; Raptor; RealSecure; Sidewinder]. Whereas ASIM/CIDDS seem to be more USAF oriented and proprietary, AIDE is a rather "open" system more centered towards heterogeneous DoD networks. CIDDS designers are currently working on a tap to allow CIDDS data to be sent to AIDE.

2.2 Object Persistence

Object-oriented (OO) systems promise to solve real problems with greater reliability, increased programmer productivity, and a greater degree of modularity within the code produced [Loomis 1]. Through the years, IDSs have mainly been coupled with relational databases to store information that would be needed for future use. Although this seems logical since relational databases have become the standard for data storage on hard media, there are many limitations and performance issues that arise when translating objects into a data format that can be used with standard SQL for manipulation by a relational database.

As an alternative, object-oriented databases were built to provide OO systems with a means of storing data persistently across executions of programs built around an OO design. The database literally becomes an extension of the OO language, so that the language itself supports the database specification and maintenance [Montgomery 81].

To be considered an OODBMS, several key features must be present. The database must provide a means of forming data items into elements such as records or objects. There must also be a way to group objects into complex data structures, such as lists, relations, sets, bags, and trees. There must be a way to query and access stored information while providing operations to create, modify, and delete data structures. And finally, as mentioned previously, there must also exist a means to store objects persistently [Montgomery 82].

2.2.1. *What is an object?*

OO programming is based on an atomic entity known as an *object*. A software object, formed from the union between state and behavior, is an independent encapsulated representation of real or virtual world things; living or non-living. As Pierre-Alain Muller points out, "The size of these objects varies a log: some are small, like grains of sand, and others are very big, like stars [17]." He also points out that objects may not have any mass. Given OO modeling and programming, software can capture all of these entities as software objects.

To be an object, there are a number of characteristics that the entity must possess:

$$\mathbf{Object = State + Behavior + Identity}$$

1. *State*. This refers to the group of *attributes* that an object contains at a given point in time. Attributes are pieces of information that qualifies the containing object [Muller 19-20]. For example, a person object may contain a name, age, height, and weight. In one state, the age may be 23. In another state, the age may be changed to 24. The change in age demonstrates one basic rule concerning states; states evolve with time.

2. *Behavior*. This refers to the abilities of an object, which translates to actions and reactions. In programming, these are referred to as *operations*. Operations are the result of a message sent from one object to another or a message sent from an object to itself [Muller 20]. To extend our person analogy above, the person may have operations such as run or walk.
3. *Identity*. Each object must somehow be identified from all other objects that contains exactly the same attributes and operations. Although it is implicit and normally relies on memory addresses or some other system-level object identifier, there are sometimes natural identifiers that are contained within each object, such as a name, SSN, or college ID number [Muller 21].

Objects also have messaging categories by which they are created, destroyed, queried, and manipulated. Messages create the fabric by which objects communicate between themselves. Below are the five main message categories [Muller 24].

1. *Constructors*: create objects
2. *Destructors*: delete objects
3. *Selectors*: return all or part of the state of an object (get methods)
4. *Modifiers*: change or manipulate all or part of the state of an object (set methods)
5. *Iterators*: visit the state of an object or access a data structure that contains a number of objects

2.2.1.1. Classes

Objects, in general, are far too complex to be understood as a whole at one time. There may be thousands or millions of objects to work with (consider the number of stars) simultaneously, so designers of systems must group various types of objects together and focus their thinking. This leads to a more abstract method of object representation, known as a *class*. A class is a description of an object without reference to state. Since many objects simply look alike and contain the same attributes, then a class may describe multiple objects. Objects, therefore, are *instances* of classes and are built during program execution through a process known as *instantiation* [Muller 30]. Classes are to objects much like schemas are to databases. When a database is built it contains a schema that describes the columns and ranges of values that those columns may take as valid arguments. Although the database may be completely empty of any actual rows containing information, the schema still exists as a description of the database. In much the same way, a class exists as a description of objects to be built, whether or not objects of that class have actually been instantiated.

Table 2-1. Graphical Representation of Class Using UML

<u>Person</u> (Class Name)
(Attributes)
name age weight height
(Operations)
run() walk() Person() - Constructor setName() - Modifier getName() - Selector

Table 2-1 shows a graphical representation of a person class using the Unified Modeling Language (UML). UML is one of the methods by which OO designers build and

model OO systems and will be used throughout this thesis to describe classes and their relationships.

2.2.1.2. Relationships

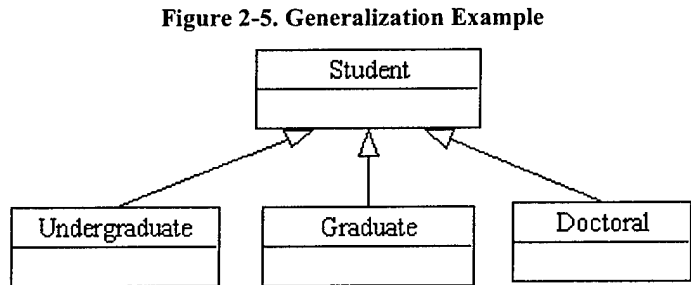
Communication must exist between objects so that messages may be passed from one object to another. In modeling classes, there is also a method to represent these links known as association. One other method, known as aggregation, is also used to represent a stronger coupling between classes [Muller 36-39].

1. *Association.* An association expresses a bi-directional, semantic connection between classes. Whereas a *link* describes the communication path between instantiated objects, an association describes the path of communication between classes. An association is independent of the classes and just reflects the connection between classes that exists in the application domain.
2. *Aggregation.* As stated previously, an aggregation is a stronger coupling between classes than that of an association. It is purely logical and in actual implementation, is not handled any differently than a normal association. As an example of an aggregation, consider a composition. A car has an engine and wheels. Although a car would be associated with both of them, that association would be considered a composition due to the fact that a car must have an engine and wheels to be considered a normal car.

2.2.1.3. Generalization and Specialization

Generalization and specialization may be implemented the same way within the object hierarchy, but the two are actually opposite concepts. Most often, inheritance is used to show classification between classes. Inheritance is discussed later in this section.

When multiple classes may be abstracted to produce another common class that contains some of their like methods and attributes, a generalization emerges. The resulting common, shared class is known as a superclass. Figure 2-5 below shows an example of generalization. [Muller 40]



In the example, each of the leaves at the bottom are types of students, and thus share a number of properties. The shared properties are migrated to the Student class, such as name, graduation year, address, etc.

By contrast, specialization involves taking a superclass and breaking it into specialized pieces. Whereas generalization is a bottom-up design decision, specialization is a top-down decision, which normally is used to provide reuse and extension of the superclass. [Muller 41]

One of the more common ways to provide generalization and specialization is through inheritance. Inheritance is a method by which classes inherit methods, attributes, and sometimes constraints from the superclass. In the above example of generalization, the Student class may contain methods to calculate GPA. All three of the subclasses may use this method and do not need to contain their own version of the method to do so. They call the method from Student as if the method actually existed in the subclass.

2.2.2. Relational vs. Object-Oriented Implementations

An RDBMS and an OODBMS differ from each other in a number of ways. Table shows some of the more fundamental differences [Blaaha 182-187].

Table 2-2. RDBMS vs. OODBMS Comparison

RDBMS	OODBMS
Data presented in tables	Data maintained in object form
Operators for manipulating tables (SQL)	Proprietary database manipulation language or ODMG standard
Constraint checking provided in database	Constraint checking provided by programmer in OO language
Slow navigation between objects (joins are costly)	Fast navigations among objects since no joins are necessary
Mature theory and standards	Immature technology (debatable)
Distribution difficult	Distribution inherent to design (dependent upon database)

2.2.2.1. Object Persistence and the RDBMS

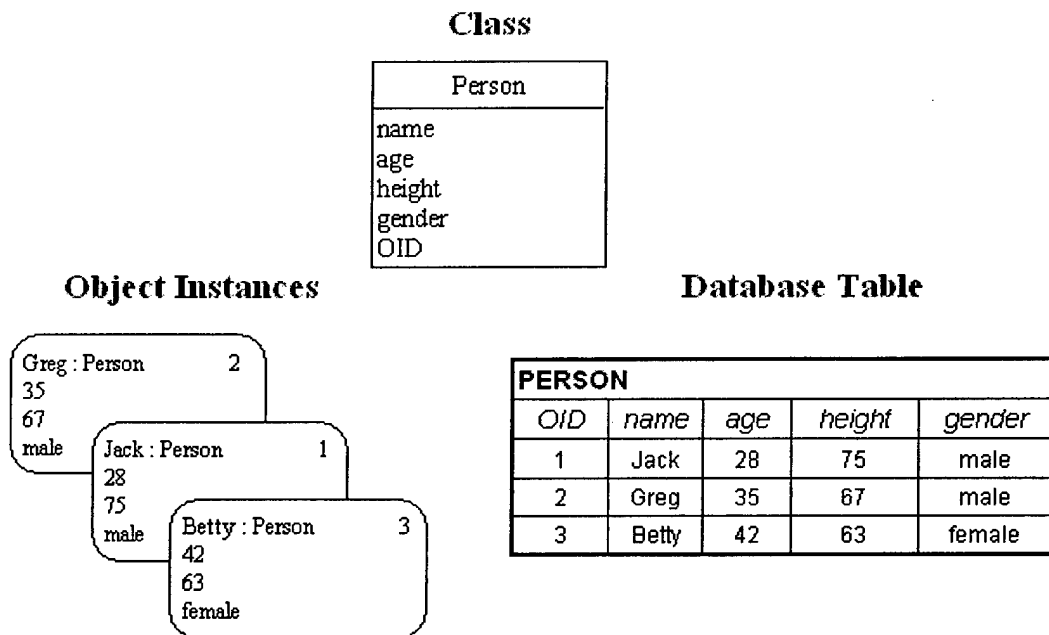
In order for information to remain persistent through executions of an OO program, a data store is necessary to hold information pertaining to the objects and relationships between them. When using an RDBMS as the storage medium, a number of techniques are used to bridge the gap between the object and relational representation of the same data.

- *Swizzling*. Object persistence using an RDBMS is achieved mainly through the definition of table structures that represent the objects to be saved. When an object needs to be written to disk or modified in some way, a method is called within the program containing the object which contains SQL that inserts, deletes, or updates the attributes of the object in its given table [Kroenke 488-485].

There are a number of concerns that arise when this process begins. First, the

object must have a unique identifier for the database to use as a primary key. When an object is in memory, it has a unique address in memory and handler within the program that identifies it from other objects. Thus, several objects can have the exact same attributes, but can always be identified by the program as being unique, individual objects. When that object is to be made persistent, the RDBMS needs its own way of identifying those objects, since no two tuples in a table can have exactly the same column values and still be considered unique. One way to handle this issue is to have the object keep a separate attribute that equates to a primary key and use this attribute as its primary key within the database. This key

Figure 2-6. Object to Relational Mapping Using OID



is often referred to as an object identifier (OID). When an object is deleted from memory, it can then still be identified from those that are in the same process

[Kroenke 489]. The process of translating a memory address to something the database finds more useful is known as *swizzling* [Kroenke 487]. Figure 2-6 shows an example of swizzling between an OO program and an RDBMS. Note that the OIDs in this case are user-defined.

- *Associations*. Since an object cannot directly be made persistent using its own address, associations between objects must also be translated in some way so they remain intact while being stored in the RDBMS. In an OO program, associations are simply memory pointers or references between objects so the program can relate one object to another. Pointers and references rely on memory addresses that remain fairly constant while the program is executing. Once a program ceases execution, the memory addresses used by its objects are generally returned to the operating system for use by other processes. When the process of swizzling is used to keep track of objects within the database, the objects must also pass their OIDs to any objects that reference them. This can create a problem when, say object_A is deleted, but object_B's association with object_A remains intact. In essence, the memory address that object_A was located at is no longer being used, but the OID located in object_B is still being used by the database to keep track of object_B's associated objects. This means that when we make even a subtle change within the process using the objects, those changes must somehow be sent to the database so the executing process and database are synchronized [Kroenke 489].

Associations between objects may also spawn new tables. When a program needs

a many-to-many relationship defined, a new table containing the foreign keys of each object must be populated. This simply adds to the space requirement of the database and causes additional updates and maintenance for administrator and programmer alike.

- *Programming Code.* The amount of code needed to make objects persistent is also an issue. Primarily, in an OO program using an RDBMS for storage, there are a number of conversions that must be performed to get the data into a relational form. The following table, Table 2-3, shows one instance of possible program flow in an OO program using object-to-relational mapping techniques. Notice that when one change or action is performed within the OO environment, certain extensions to the program are necessary to accommodate the RDBMS and keep the database itself current.

Table 2-3. Extensions Needed When Utilizing an RDBMS

Object-Oriented Application	RDBMS Extensions
Execution begins.	Connection to RDBMS made. Tables built (if necessary).
Objects instantiated and populated.	OIDs created within objects for database identification.
Associations made between objects (pointers and references).	OIDs passed between associated objects.
Objects save method called.	Save method creates SQL INSERT statement, which maps the object to appropriate table and columns.
Associated objects' save methods called (recursively).	Again, save method produces SQL statement, but uses parent OID as a foreign key to maintain association.
Objects are modified. Update method called.	SQL UPDATE statement used to update appropriate fields within record.
Objects are deleted. Delete method called. Its association to other objects is NULL.	SQL DELETE statement deletes record from database just prior to object actually being deleted. Additionally, associations are found and deleted from database (lookups).
Program terminates.	Database maintains objects.
Execution begins.	Connection made.

Table 2-3. Extensions Needed When Utilizing an RDBMS

Object-Oriented Application	RDBMS Extensions
Objects instantiated. Read method called.	Attributes are read from database, also placing OIDs into objects.
Associated objects instantiated (recursive). Read method called in each.	Attributes read from database, and OIDs are passed between objects. If a to-many association exists, OIDs must be passed to objects on both ends of an association.
Execution Continues	

The resulting situation is that the amount of code needed to keep an object persistent throughout the object's lifetime becomes a substantial part of the program in general. The additional code can cause program bloat that may ultimately result in more programming bugs, slower performance due to overhead, and an increasingly large amount of maintenance in keeping the program SQL statements current with the database schema. Put simply, RDBMS implementations generally integrate poorly with OO languages [Blaha 184].

- *Strengths.* The RDBMS does have its strengths. However, these strengths do not apply to its implementation of object persistence. The technology surrounding the RDBMS is fairly mature and advanced. As a result, it is generally considered more stable. The RDBMS is also more business-rule oriented and is well-suited for day-to-day transaction-based systems centering on normal business logic. Finally, they are widely available and support is easily accessible [Blaha 184].

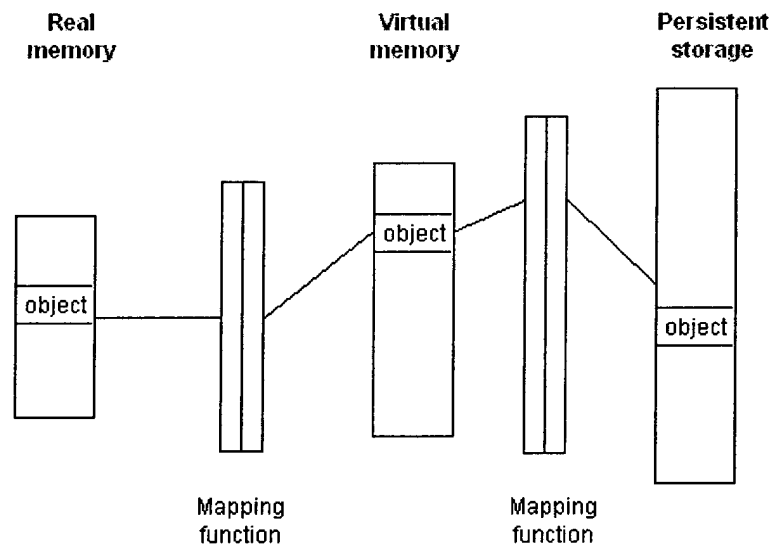
2.2.2.2. *Object Persistence and the OODBMS.*

Where relational databases are weak in their implementation of object persistence, the OODBMS is strong. It should be noted, however, that implementations concerning almost all areas of the OODBMS are vendor-dependent. Some are naturally

distributed due to their particular use of OIDs, but most are not. Some interface with C++, Java, and Smalltalk, while others pick and choose the languages that they will work with. Of course, this is also true in evaluating performance and storage needs between database implementations. Although Objectivity/DB was chosen as the database of choice for this project, characteristics that are common to most OODBMS implementations will be discussed in this section as a general overview of the technology. Objectivity/DB specific issues are discussed later in this chapter.

- *Natural Language Extensions.* Unlike the RDBMS, an OODBMS is built around one or more OO languages to integrate almost seamlessly into the native programming environment. This means that there is no mapping mechanism needed to parse an object and its associations into pieces for insertion into a relational database. Instead, the persistence is built and compiled into the object and the database that resides on hard media (i.e. hard disks) is treated as another source of memory. The Figure 2-7 below shows the memory hierarchy.

Figure 2-7. Memory Hierarchy using OODBMS



Moving objects between system memory and persistent storage is almost completely transparent to the programmer. Objects are declared as persistent and when they are created or modified, the changes are immediately sent to the database with little programmer direction using an internally maintained OID. The exception being that normally a transaction of some type must *wrap* the object when it is used so that the executing program knows to check the database for the most up-to-date copy of the object. Associations are handled in the same manner. Associations are declared ahead of time as links between objects. When the programmer finally assigns objects to each end of the association, the OIDs of the objects are placed in the database. The database and the executing program handle the OIDs; at no point does the programmer define an OID for any object [Loomis 124-125].

- *Link Navigation.* Traversal of the associations between objects is really where the OODBMS gets its speed with highly complex data sets. In the RDBMS, joining tables is the standard method of pulling information. The process is slow, tedious, and can consume a large amount of resources depending upon the tables' sizes and composition [Blaha 183]. In an OODBMS, associations are stored with the objects and there is no need to join tables to get information. Instead, the database follows the pointer (OID) to the associated object. An RDBMS is well-suited for retrieving and inserting information from a database when a small number of tables are involved and joins are limited [Blaha 183], but once the data set and number of tables gets large, the OODBMS method of storing OIDs with the objects becomes much more efficient.

- *User-Defined Data Types.* As mentioned previously in Chapter I, complex and user-defined data types are often needed to encapsulate and define information. When
- *Schema Evolution.* It may be necessary to change the schema in a major fashion to properly represent the data. From a programmer's perspective, this means that the object-to-relational mapping code would need to be revamped as well. In an OODBMS, the database schema would only need to change if the objects in the programming environment also needed to change. There are usually special schema evolution tools to handle this conversion, thus taking the burden of modifying a large number of SQL statements off of the programmer. As an example, Objectivity/DB contains a schema manager that automatically checks the database against the objects that are currently in use and can be set to change the database when differences are found, such as added fields, field name changes, added or changed associations and added or deleted classes. Absolutely no intervention by the programmer is necessary.

2.2.2.3. OODBMS in Practice (Objectivity/DB)

As previously stated, an RDBMS is well suited for small data sets where table joins are at a minimum or where computations may take place over a large number of records. Such systems could be thought of as simple transaction-based systems. When the schema becomes very large, it is only natural for the number of associations between tables to increase, or even the addition of new tables to handle many-to-many relationships. Complex data sets are the primary reason why the Stanford Linear Accelerator Center (SLAC) chose to use Objectivity/DB to store event data from its high-energy physics experiments.

SLAC is an experimentation network of 800 physicists collaborating from more than 80 sites in 10 countries. Its goal is to use big-bang energies to create B meson particles, look at collision decay products, and then find the anti-matter that should be present. In all, the experiment is expected to produce between 200 - 400 TBytes of data/year for 10 years [Hanushevsky 2].

Initial plans were to include the use of a relational database as the main storage mechanism. However, using tests from previous experiments, it was found that the object-oriented approach outperformed the relational approach by factors ranging from 5 to 50. These numbers were significant to the project, and thus an OODBMS is now being used for data storage [Meyer 1-2].

Objectivity/DB itself was chosen for a number of reasons.

- It accommodates databases greater than 100TB in size.
- Includes optimized protocol for distribution (AMS) that is more efficient than conventional network file systems (NFS).
- Uses page-based client/server architecture and memory cache to allow large numbers of objects to be accessed within a single transaction.
- It was found to scale better than many of its competitors to large numbers of objects because of its memory cache (vice virtual memory).

The use of Objectivity/DB at SLAC is of great importance given the perceptions of many database administrators that OODBMS technology is still too immature for large projects [Kroenke 492]

2.3 RDBMS to OODBMS Schema Transformation

In some cases, it may be necessary to translate a schema which was originally developed for use in an RDBMS into one which functions well in a complete OO environment and is used in an OODBMS. This would involve taking the database, tables, and columns and converting them into containers, classes, and attributes (fields), respectively.

Such procedures are well understood in the OO community as OO programming becomes more prominent. A thesis by Pedro A. Linhares Lima from AFIT describes seven steps in translating a relational model to an object model; the steps are given below. These steps, with a few additional procedures, are also described in more detail by Michael Blaha et al [Blaha 451-454]. For this thesis, Linhares Lima's method is adequate:

- (a) *Prepare an entity-relationship (ER) model.* Each table is represented as an entity where candidate, primary, and foreign keys are determined based on existing database [Lima 17-20].
- (b) *Prepare an initial object model.* Each entity in the ER diagram is represented as a tentative class and each relationship as a tentative association. All columns within the tables become attributes of classes [Lima 20].
- (c) *Refine tentative classes.* Combine classes that have the same schema into a single class. Take out classes that were represented in tables for function and constraint purposes. These classes normally do not participate in any foreign key, and therefore should not have any associations [Lima 20].
- (d) *Discover generalizations.* Look for large foreign-key groups and primary keys composed entirely of a foreign key from another table. Generalizations may also be seen where there are patterns of many replicated attributes and where

there are patterns of data where a class has mutually exclusive subsets of attributes. In many cases, attributes were pushed up a class or down a class to conform to tables used in relational database since an RDBMS cannot entirely depict inheritance [Lima 21].

- (e) *Discover associations.* Convert tentative classes to associations when two or more foreign keys make up the classes in question. Determine associations based on distribution of foreign keys throughout classes and state maximum and minimum multiplicities for each. Some associations will become aggregations, where a class is “a-part-of” another class [Lima 21-23].
- (f) *Perform transformation.* Look for transformations that were made to the original relational database for the sake of improving time and/or space performance. Some of these transformations involve transitive closure, lightweight one-to-one associations, and the combining of associations and generalizations [Lima 24-26].
- (g) *Prepare a functional model.* This model describes the computations within a system. In a database system, this model is trivial given that its primary purpose is to store and organize data. Preparing the functional model is achieved through studying the user manual and forms and possibly interviewing users [Lima 26].

2.4 Federated OODBMS

In the case of the AIDE database to be used in the experimentation portion of this thesis, one of the main goals is to distribute the database across multiple sites to both aid in

the collection of information from multiple sensors and allow updates to information common to all the sites. The intention is to (a) divide up the information domain so as to not tax any particular machine or portion of the overall system with collecting more information than it can keep up with and (b) to be able to make changes to one portion of the system and have those changes automatically distributed across the system's domain of sites. To do so involves distributing a databases system using multiple component DBMSs.

2.4.1. Distributed DBMS (DDBMS)

M. Tamer Ozsu et al present one definition of the DDBMS [Ozsu 4]:

We can define a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the DDBS [Distributed Database System] and makes the distribution transparent to the users.

A DDBMS is essentially a set of component databases joined together via a DBMS architecture that allows all the component DBMSs to communicate and function as a single database system. The resulting DDBMS provides users access to stored data as if the database system was a simple, centralized store where data is kept in a single database. The fact that multiple DBMSs are actually participating in the storing of the data is completely transparent to the end user. Note that a network is involved in a true DDBMS.

There are a number of advantages that DDBMSs are expected to provide:

1. *Transparent management of distributed and replicated data.* Transparent management refers to the concept of hiding all implementation details from the end user. Replicated data is data that exists at many different sites within the distributed system. In providing these two main features, transparency and replication, the DDBMS must provide data independence, network transparency, replication trans-

parency, and fragmentation transparency [Ozsu 8-12].

In short, the DBMS must be able to provide the user with data without the user knowing:

- (a) What DBMS actually stores the data
- (b) How the data came across the network to its final location
- (c) If the data is located relatively close to the user's machine or if it exists at some rather remote site
- (d) Whether all the data that comprises the end result was located in the same location or put together from fragmented sources and then assembled

2. *Reliability through distributed transactions.* When DDBMSs distribute their data across a network, then the single point-of-failure issue is eliminated. However, there must be a means by which users can access all the data in the database transparently, as mentioned above in (a). A transaction consists of multiple database operations executed as a single atomic action. When a transaction is submitted to the database, it is assumed that although there are concurrent transactions taking place, the database will be able to return a correct result without violating database consistency. A transaction used in any database context has the following proper-

ties, known as ACID [Larson 120]:

Table 2-4. A C I D

Atomicity	Either all or none of the transaction's operations are performed
Consistency	This follows if all the data in the database satisfy a set of business rules both before and after a transaction transforms the database
Isolated	Concurrent transactions' operations are insulated from one another
Durability	When committed, results are persistent

In a distributed architecture, the problem is amplified due to the fact that multiple DBMSs are involved. Therefore, a single *logical database image* exists that users access in order to find whatever data is needed. Once the image is accessed, it is up to the DDBMS to locate the applicable data for a given transaction and return a result regardless of failures within the system or concurrency issues [Ozsu 15-16].

3. *Improved performance.* Performance increases are based on the notions that the database can be fragmented or data can be replicated across multiple sites. In fragmentation, that data is split into pieces so frequently used data is located relatively close to an application that uses it. Also, applications can be run in parallel either accessing the same data that has been replicated close to the application or breaking up queries to access data located at multiple sites [Ozsu 17].
4. *Easier system expansion.* Systems can increase productivity by adding additional computers that contain distributed data. Instead of replacing an entire database system to handle additional needs, systems can just be added to the existing architecture.

2.4.2. Distributed Object-Oriented DBMS (DOODBMS) Design

Designing a database around a DDBMS is much more complex than when designing around a centralized database. Due to concurrency, transparency, and replication issues, a designer must take into account how the system is to be laid out across multiple sites to achieve maximum performance and concurrency. Add to this mix object representations consisting of relationships and inheritance that build hierarchies that can span across multiple databases, and the problem simply increases in size.

There are a number of factors to consider when deciding to distribute an OODBMS. In his thesis, Capt Hsin-feng (Edward) Wu identifies two major considerations mentioned above with respect to performance that need to be taken into account when a DOODBMS is being used; data fragmentation and replication [Wu 2-48 - 2-55]. James A. Larson also identifies these considerations and gives three simple ways to approach the problems inherent to splitting data amongst multiple databases. This thesis will concern itself with fragmenting and replicating objects.

2.4.2.1. Fragmentation

Fragmentation involves partitioning a set of data into multiple fragments for storage in different databases. Although Larson speaks from a relational point-of-view with rows and columns as the main storage method, the same fragmentation concepts apply to the object-oriented paradigm using objects and attributes (or fields in Java). There are three methods by which to fragment data in a database; horizontally, vertically, and mixed [Larson 83-87; Wu 2-48 - 2-51].

In horizontal fragmentation, the database table's rows are partitioned so that independent sets of data are separated by rows. In essence, the fragmentation approach would place rows into tables where they would most likely all be accessed together, thus eliminating the need to query rows that are not to be included in any of an application's queries.

In vertical fragmentation, columns are separated into multiple tables so that those applications that do not regularly need a number of columns' data will return a new table containing only the data that is needed. For instance, if there are four columns in a table and an application or set of users regularly only needs access to two of them, it may make sense to partition the table into two tables with two columns apiece.

In mixed fragmentation, a table is partitioned using the horizontal and vertical methods together.

2.4.2.2. Replication

Replicating data involves making multiple copies of a piece of data and distributing those copies to multiple databases. This may be done for two reasons; to provide greater availability to the data and to reduce communication costs by allowing the data to be located closer to an accessing application [Larson 134-135].

In an OODBMS, as was discussed earlier, replication can be done with logical images of databases. An image allows a user to update one copy of that database and have all the other copies update automatically.

2.4.3. Tightly-coupled vs Loosely-coupled DDBMS

There are two main types of distributed databases, tightly-coupled and loosely-coupled. In the former, a database has a controlling entity, the DDBMS, that maintains business rules for the entire multi-component database system. All requests must be forwarded through the DBMS before proceeding onto those portions where the data is stored [Larson 11-13]. This type of database system also performs all the necessary optimizations, translations and data merging during a request.

By contrast, a loosely-coupled DDBMS allows users to directly access data in several component DBMSs. Therefore, no global business rules are in place to protect the entire system's component databases. Such rules must be established before a request

reaches a component DBMS. The loosely-coupled system also allows for all the functionality of the tightly-coupled DBMS with the exception of global rules.

2.4.4. Federated DBMS

A federated DBMS (FDBMS) is a loosely-coupled DDBMS that allows for all the functionality of a DDBMS, but users may access any component DBMS without using the distributed features built-in to the overall system. This means that a program can access data physically located on its own system without ever invoking inter-database communication mechanisms. Thus, data can be controlled and accessed locally, yet still participate in the overall distributed database architecture when remote data is needed [Larson 45].

A federation is comprised of multiple databases that participate in the federation over a network and a *federation schema*, which consists of *export schemas*. The export schema describes all of the data that users can access within the component DBMSs. A federation schema describes all the data in the export schemas of participating component DBMSs. For example, an administrator may only grant a set of users with permissions to select certain pieces of information from a component database; all other information is to remain out of the users' reach. The administrator would build an export schema that describes the data that the users may access. This export schema, along with others used throughout the system, are actually part of one, large integrated schema that describes the entire federation. This larger schema is, in fact, the federation schema [Larson 47].

2.4.5. DOODBMS Design

Modeling a DOODBMS is not much different than modeling any other existing OO system. There still exist the notions of analysis, design, system development and integration. However, because many of the components are built to be distributed, there are additional considerations that need to be addressed. These include data dictionaries for system definitions, services, collections, containers, and object placement within the system.

Wu, mentioned in "Fragmentation" starting on page 43, established a set of guidelines for designing a DOODBMS. The basic definitions of the seven activities that he produced are presented here:

1. *Object-Oriented Analysis (OOA)*. In Wu's first step, he starts with OOA to determine information concerning the database, application, and user access patterns.

For the most part, this step concerns itself with designing the objects to be used in the system and the relationships and inheritance between objects just as in any OO application development environment [Wu 3-4]. More specifically:

- (a) Identify classes and objects
- (b) Identify object and class structures
- (c) Identify object attributes (fields)
- (d) Identify the subject for each abstraction
- (e) Identify services between objects
- (f) Define data dictionary for classes, services and attributes
- (g) Test the design
- (h) Apply inheritance where appropriate

2. *Perform high-level design*. In this step, Wu transforms the OOA model into an actual database design. This involves the following:

- (a) Take the results from OOA and treat as a conceptual design.

- Identify design trade-offs through system design
 - Identify superclasses and subclasses
 - Resolve many-to-many and one-to-many relationships with respect to how they will be represented in the database
 - Apply inheritance
 - Refine generalization, specialization, aggregation, and association to better model data
 - Design problem domain, human interaction, task management, and data management components
- (b) Design global conceptual schema by combining OOA and OOD into the overall application database design
- (c) Distribution design: partition objects according to responsibility and user view and consider network partition
- (d) Determine object placement considering responsibility and cost and performance and consider trade-offs of placing objects into distributed global database
- (e) Design local conceptual schemas by treating each object as a unit of distribution and perform low-level design of objects as described in next section

3. *Perform low-level design.* The purpose of this step is to concentrate on implementing the design given the language to be used [Wu 3-5].
 - (a) Determine object representation (class, data type, or static object)
 - (b) Refine inheritance structure
 - (c) Implement object methods with respect to design strategies
 - (d) Establish object visibility (private, protected, public) to establish communication between objects
 - (e) Identify polymorphic methods
 - (f) Apply OO design (OOD) using programming language to binded language
 - (g) Perform physical design and map the local schemas to physical storage devices.

4. *Select OODBMS.* In selecting the particular OODBMS to be used, Wu suggests performing a survey of commercial OODBMS platforms and select one suitable to the needs of the organization and application [Wu 3-34]. The following criteria are used for evaluating an OODBMS:
 - (a) *Power.* Determine whether or not the database will support full distribution and OO concepts. Next, measure performance and response of the system.
 - (b) *Ease of use.* In determining ease of use, consider user interface and learning curve.

- (c) *Robustness*. Determine if platform is compatible between versions and if it is reasonably “bug-free.”
 - (d) *Functionality*. Consider whether or not the OODBMS fits the adopted methodology, as well as, the specific functionality of OODBMS (i.e., language interface, schema evolution, lock management, communications, concurrency, etc.).
 - (e) *Ease of insertion*. Determine if the OODBMS is available for the organization’s chosen hardware platforms and whether or not the installation instructions are precise and clear.
 - (f) *Quality of support*. Determine the level of maintenance support and hotline service provided.
 - (g) *Others*. Consider future development, cost, GUI standards, reputation, portability, etc.
5. *Determine computer aided software engineering (CASE) tools*. Wu’s thesis also deals with the selection of CASE tools to aid the designer in building the applications necessary to administer, maintain and build the OODBMS [Wu 3-36]. This step is listed in Wu’s thesis as part of step 4 above. However, it has been separated here for clarity.

6. *Code using OO language.* Apply OO methodologies and principles to map classes, objects and structures to language of choice or code generators for generation of OO database schema. Next, apply methods of structured programming [Wu 3-37].

7. *Observe and monitor the application.* Build the applications and monitor its performance. Refine as needed and improve stability given user input [Wu 3-37].

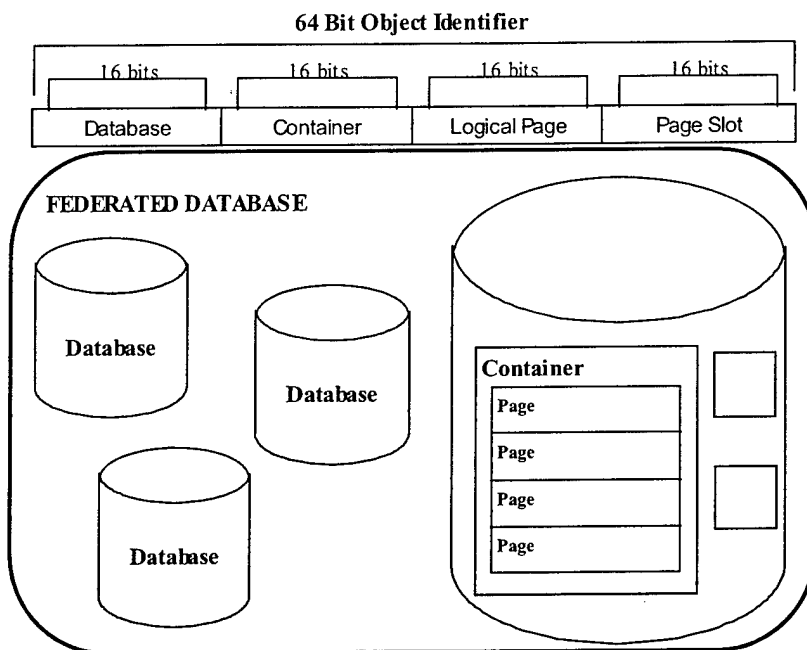
The above activities are merely a brief overview and much more guidance is given in the respective thesis as to how to implement each and every step. The more detailed breakdown will be presented in Chapter IV.

2.5 Objectivity/DB Overview

2.5.1. Architecture

Objectivity/DB is robust Federated OODBMS, which handles persistent objects in a heterogeneous environment. Coupled with Java, C++, or Smalltalk, Objectivity/DB allows seamless integration into the OOP language for use of objects regardless of which language built the database or stored the data being accessed.

Figure 2-8. Objectivity/DB OID Composition



To achieve the distribution across networks and allow for uniqueness among objects that may have the same attributes, Objectivity/DB uses a 64-bit OID composed of four 16-bit fields. Figure 2-8 shows the actual OID breakout [Objectivity/C++ 1-3].

As seen in the OID, there is a common hierarchy of storage classes that an object must belong to [Objectivity/C++ 1-2].

- *Basic Object.* A fundamental unit of storage consisting of scalar types (char, int, float), structures and class instances, strings, array types, associations, and object references. Each object is contained within a container.
- *Container* - collection of basic objects. Objects within a container are physically clustered together in pages on disk for efficiency. *Database locking occurs at the container level*, which is a very important consideration when designing where database objects will be stored. Each container belongs to a database.

- *Database* - collection of containers. Contains a default system container and user-defined containers. Databases are stored as individual files on disk. Each database belongs to a federated database.
- *Federated Database*. Logically contains databases and schema describing all publicly visible class definitions. This is the highest level and where administrative control is achieved.

Objectivity/DB also provides its own lock server (LS) and advanced multi-threaded server (AMS) that perform functions vital to concurrency and distribution. Database access by multiple users is achieved through locks administered by the LS. A single LS is used for each federated database or autonomous partition. A database, container, or basic object may be opened in either read or update mode. When an object is opened, the lock actually exists at the container level. The lock server maintains the lock until the transaction is committed [Objectivity/C++ 11-2]. However, a concurrency mechanism known as Multiple Readers, One Writer (MROW) is available that allows multiple process to read data even when a container is being written to. MROW occurs at the transaction level, which means that the reader receives the last good known version of the data (before the current updating transaction that holds the lock), and significantly improves concurrency [Objectivity/C++ 11-8].

AMS provides a method of distribution by allowing databases within a federation to request data from any other database. Both the LS and AMS open a port on the server for communication purposes.

2.5.2. *Development*

Developing an application using Objectivity/DB is not unlike developing a normal C++ application. However, when using Objectivity/DB types within the program and com-

piling the application with the database data definition language (DDL) file, objects declared as persistent are created within the database and persistence is achieved upon execution of the main program. The steps below are used to develop the application [Objectivity/C++ 1-6].

1. Schema Development

- (a) *Prepare DDL and identify classes to be made persistent.* In essence, the DDL file replaces the normal header file normally associated with C++ applications.
- (b) *Create federated database (if necessary) using oonewfd.*
- (c) *Process DDL file using DDL processor.* A header file is automatically produced containing normal class definitions and additional Objectivity/DB persistent methods and attributes.

2. Application Development (C++)

- (d) *Produce application code based on schema.* Include the DDL and header files in application source code.
- (e) *Set up environment including the boot file.* The boot file contains information pertaining to the lock server and federated database disk location.
- (f) *Create or run Makefile which:*
 - Runs DDL processor
 - Compiles application

- Links with Objectivity/DB libraries and object files

(g) *Build and run application.*

(h) *Use ootoolmgr to view database data.* This tool allows the administrator or creator to view and manipulate any data contained in federation, database, container, or basic object.

3. *Rational Rose Link.* A tool called Rational Rose Link is available to automate the creation of the DDL file and federated database for the C++ environment. Rational Rose is a development environment using a graphical interface to create classes, associations, attributes, and many other constructs used in the creation of Java and C++ programs. Using the link, an object diagram can be created graphically and each class can be made either persistent or transient. Transient classes are those that are destroyed after the program has terminated. Associations are also maintained by the interface along with attributes and any needed methods, including sets, gets, and constructors. After the object diagram is completed, the DDL and federated database are automatically created for the user.

4. *Application Development (Java)*

With Java, the header files and DDL files are eliminated. The only stipulation is that if a basic object is to be read, the class reading it must contain exactly the same attributes. In creating the database, the class must extend the appropriate Objectivity/DB persistence class.

2.5.3. Transactions

Transactions are the method by which data is modified or inserted into a database. When a program is first executed and database data is to be accessed, the program declares a transaction. After a transaction is started, the database can be used. Any methods dealing with the database must be started and ended within a database transaction or the data created or manipulated will not be available for others to access. A transaction will continue until it is either aborted or committed using the abort, commit, or commitAndHold functions [Objectivity/C++ 2-7].

Since a transaction is not committed until commit or commitAndHold are explicitly called, journal files may be used when the transaction aborts, either explicitly or by default if an error occurs. Journal files are maintained along with the other federated database files, which contain information concerning recent uncommitted transactions. The database performs a roll back of the data and returns the database to a consistent state as it was before the transaction took place. Roll back can occur when an application, client host, or lock server fails. A manual method to recovering the database also exists if the database host becomes permanently unavailable after a failure [Objectivity/Admin 113-114].

Concurrent transaction support is also supported. This enables a user to run multiple, concurrent transactions over the same federated database from within the same process. Each transaction is started using its own non-preemptive thread, which means they will not be switched unless the user explicitly switches between them. The only drawback seems to be that you cannot use the Objectivity/DB threading support in conjunction with a threaded package that uses preemptive scheduling. This is due to the fact that each Objectivity/DB thread contains session/transaction information including, among other things, its own copy of the buffer cache and error handlers [Objectivity/C++ 2-18].

2.5.4. Security

Security of the database is handled using the operating system's file system. Permissions are set on the federated database, autonomous partition, database, journal, and boot files so as to let those that need access the ability to open and close the database for routine usage. The LS and AMS components must be run from an account that has access to any files related to the databases that the two components control [Objectivity/Admin 29].

2.5.5. Distribution

Distribution is achieved through another layer of data abstraction known as an *autonomous partition*. Partitions are completely self-supporting and can contain and manage multiple databases and containers. Self-supporting means that the database has its own boot file, lock server, and system database file, which contains schema information and the global catalog of partitions along with their locations and a listing of all the databases they contain [Objectivity/Java 274]. Using the autonomous partitions, Objectivity/DB provides a Fault Tolerance Option (FTO) and Data Replication Option (DRO). In distributing the data using these two mechanisms, two goals are achieved.

1. *Fault tolerance*. With FTO, multiple users are able to access data regardless of failures outside of the partition. This is because the partition maintains its own lock server and database schema, which are normally only located at the federated database level [Objectivity/FTP-DRO 16]. Put simply, if the rest of the federation is unavailable, work may still continue in the working partition.
2. *Data availability and performance*. Data replication through Objectivity/DB DRO extends the FTO and allows a database to be replicated across multiple partitions on the network using *database images*. Images of a database are created and dis-

tributed to multiple partitions and when a quorum, or a majority, exists amongst all the images, and then data is written out to those images [Objectivity/FTO-DRO 23]. However, data can be read from any image regardless of whether or not a quorum exists. This arrangement allows quick access to data, but keeps the maximum number of available images updated. This feature can be considered the backbone of Objectivity's distributed architecture. The DRO alleviates the single point of failure problem that exists in a centrally located database architecture by allowing data to be manipulated in another partition if the partition that the data was originally intended for is unavailable. When the previously unavailable partition comes back online, its images are resynchronized with all the others in the federation.

2.5.6. Schema Evolution

This component, also one of the great features of Objectivity/DB, allows a database schema to automatically adjust to any changes in persistent class structure [Objectivity/Java 32]. For example, if an attribute's name is changed, the schema manager will detect the changes upon program execution and will automatically begin converting objects as they are accessed to the new class definition. In a relational database, the change would mean changing a column name and then locating and changing that name within the application. In Objectivity/DB, the process is reversed. The program containing persistent classes can change first and allow the change to filter through the database either implicitly or explicitly.

III. Methodology

3.1 Introduction

The purpose of this chapter is to describe the analysis, design, and implementation of a high-speed multiple-IDS data repository using a distributed object-oriented database management system (DOODBMS). The process begins by outlining the design process of building a DOODBMS, and incorporates a method for translating a relational database schema into an object-oriented database schema.

3.2 DOODBMS Design

The methodology for deploying the DOODBMS is nearly identical to that of Capt Hsin-feng Wu's thesis [Wu 3-1 - 3-2]. In it, Wu gives seven activities necessary for DOODBMS application design. These seven activities are presented in Chapter II under "DOODBMS Design" starting on page 45. Since Wu's design concepts primarily involve implementing a totally new design without any existing databases, all seven steps are used with a few modifications. Most of the modifications involve re-engineering an existing relational database to an object-oriented database. The method of database transformation is derived from Linhares Lima's thesis as given in Chapter II under "RDBMS to OODBMS Schema Transformation" starting on page 38 [Lima 21 - 26].

Below is an outline of Wu's steps integrated with Linhares Lima's re-engineering technique. Other exceptions and/or modifications to Wu's development technique are as noted:

1. *Object-Oriented Analysis (OOA)*. In Wu's first step, he starts with OOA to determine information concerning the database, application, and user access patterns.

This step also includes designing the objects to be used in the system and estab-

lishing relationships and inheritance between objects [Wu 3-4].

Since a system and database already exist, this first step is modified to involve mapping out the existing system and determining dataflow and components.

Therefore, there are two routes to take when approaching the initial phase of design. The first is to design from the ground up and begin the OOA just as described in Wu's thesis. The other is to begin the OOA with an existing database schema and re-engineer it to accommodate the DOODBMS.

2. *Perform high-level design.* In this step, the DOODBMS is designed using the object model resulting from the re-engineered relational database. The high-level design process includes overall system design, re-evaluation of the object model for inheritance, aggregations, and relationships, distribution design, global and local schema design, and object placement within the database. Partitioning, fragmentation and replication are also addressed in this step, which play an important role in determining how the DOODBMS is to be designed.

To complete the high-level design process, Objectivity/DB databases and containers are used along with Objectivity/DB's fault tolerance option (FTO) and database replication option (DRO). The object model resulting from the re-engineering process is mapped to the Objectivity/DB specific design structures to produce the overall DOODBMS design.

3. *Perform low-level design.* When the high-level design is completed, a low-level design process is applied that takes the DOODBMS design and implements it using the OO programming language of choice. Although this step precedes the OODBMS selection process in the next step, many OO factors may be determined to include object visibility, polymorphic methods and object methods.
4. *Select OODBMS.* Wu next advises choosing a DOODBMS based on criteria summarized in "Select OODBMS" on page 48. Objectivity/DB was selected because it meets each criterion set forth by Wu given the results of the SLAC initiative and the database implementation documentation provided by Objectivity, Inc.
5. *Determine computer aided software engineering (CASE) tools.* As described in Chapter II, Wu's thesis also examines the use of CASE tools in developing the database and its components. Objectivity, Inc. provides the Rational Rose Link, which aids in the design, implementation and testing of databases using the Objectivity/DB. Rational Rose is a CASE tool with graphic design features capable of handling CASE tool requirements defined in Wu's thesis.
6. *Code using OO language.* Once the DOODBMS platform has been chosen, coding can begin using the object model and OO programming language of choice. Java is used in Chapter IV to program the object model.
7. *Observe and monitor the application.* This, as described earlier, involves testing the application and its components, and making adjustments based on feedback from such tests. If the DOODBMS is being built from an existing RDBMS, testing the new design will need to include a comparison test against the current

RDBMS to measure any gains or losses in performance. One of the main focuses of this thesis is to provide information on performance gains or losses resulting from such a transformation. There are also two other factors that should be considered in future testing: network I/O and CPU load [Ozsu 228].

I/O is important because there may be many taps communicating with the database. In the cases of ISS' Real Secure and Network Flight Recorder, machines that collect network intrusion information rely on a set of distributed sensors to collect information and send it back to a main console where the information is stored in a database or log [RealSecure; NFR]. The taps, sensors, and console applications communicating with their respective sensors are relying on bandwidth that may already be heavily utilized by everyday, unrelated traffic. Additionally, analysts' applications need to access information across multiple networks in an attempt to gather all the necessary information to determine threat.

CPU load becomes a factor in that the taps are used on machines already deployed on a network to gather sensor and intrusion related information. The deployed tap should take up as little CPU and memory resources as possible on the machine on which it is running so that normal operations are not interrupted. This becomes a major concern in transforming a system with a centralized RDBMS because functionality is being pushed down from the RDBMS triggers and server-side applications into taps that communicate with a DOODBMS.

(a) *RDBMS vs DOODBMS performance.* If a transformation is made from an RDBMS to a DOODBMS, then it may be necessary to quantify any performance increase or decrease seen as a result of the switch. A variety of queries are made on each database and the times to return results noted. Below is a testing technique used by Tomaz Domajnko et al to measure several different methods of implementing a persistent datastore [Domajnko]:

- *Simple object insertion.* The first test is to insert a large group of simple objects into both database systems and measure the time taken by both. The objects, characterized as simple because they are not connected to any other object, are merely instantiated to create a database entry. Therefore, no additional overhead is realized as a result of placing the object into the database.
- *Complex object insertion.* A series of complex objects are placed in each database. Complex is defined as a series of simple objects with relationships made between them. Thus, not only is the overhead of instantiation measured, but also the overhead of connecting the objects into an object graph.
- *Object queries.* Objects are pulled from a relatively large database (e.g., Domajnko et al used a database filled with 45,000 objects) through a series of navigation paths. The query results are characterized by how many other objects were navigated to retrieve the desired objects and how long it took for each query to complete. The reason for this lies in the fact that a

relational database may do better on small queries involving only a couple of tables. However, as the navigation path grows larger, the time to retrieve an object is expected to increase substantially faster for the RDBMS than for the OODBMS. This is due to the fact the table joins are very costly in an RDBMS.

(b) *I/O*. The load that the DOODBMS implementation puts on a network is monitored and the application adjusted if it is found that the network is being overwhelmed by either the taps' or analysts' applications. There are a number of factors that should be taken into account when conducting measurements:

- Additional taps will only increase traffic. Therefore, measurements are made so that an estimate is made as to the expected increase in traffic given a number of taps.
- GUI applications' network load varies according to query. In a query, some DOODBMSs return pages or blocks instead of individual objects. These pages and blocks normally cluster objects together that have some logical connection. The clustering method may be defined by an administrator when the database is built or by the database itself based on relationships among data. This means that if queries by an analyst's application are made during testing that only involve like-items, then pages of those like-items will most likely be returned and cached which will mean fewer queries for objects. Therefore, it is recommended that a variety of queries be made to test the network load that an application will put on a network.

These should include queries for like-objects that are clustered together in the database and queries for completely unrelated and non-clustered objects.

- (c) *CPU load.* CPU load is monitored while the tap is running on the sensor platform. The load is measured both under controlled test conditions and during periods of peak usage when the sensor is monitoring or gathering a large amount of traffic.

CPU load and I/O will not be measured in any formal sense, but will be noted as informal observations during the testing process. These two measurements are not performed because they do not pertain directly to the objective in this thesis, which is to measure increases in performance in terms of database insertions. Chapter V contains testing results and analysis.

3.3 Database Integration into IDS Application

In 3.2, only the database itself is discussed. However, when the RDBMS is replaced with the DOODBMS, a number of key components need to be designed or redesigned to gain the full benefits of the DOODBMS. At the same time, strict coding procedures are put into effect to delegate how a program should be written to logically separate the GUI, database, and application code. In OO programming, the breakdown between portions of code is directly related to the three-tiered architecture that defines the breakdown of components related to remote database connectivity [Farley 194]. The components described below apply to the DOODBMS-based IDS data repository regardless of whether or not it was previously designed around a relational database.

3.3.1. Components

1. *Taps/Bridges.* Bridges disappear from the IDS model. Bridges mainly act as a conversion mechanism for inserting information into the database. Since the DOODBMS eliminates the conversion of information produced by a tap to SQL, the database insertion functionality is pushed down to the taps, which reside on machines running or managing the sensors.

A tap gains a majority of its bridge's functionality. Since the DOODBMS normally contains its own connection mechanism and distribution method, all occurrences where the tap used its own language or system-specific database connectivity method are eliminated. Such connection methods include Java database connectivity (JDBC), the standard by which Java connects to an RDBMS to manipulate data, and/or the network file system (NFS), the connection mechanism used to connect UNIX platforms over networks. All that is needed once the DOODBMS is installed is a tap application that performs the following:

- Creates a connection to the distributed database via the DOODBMS connection mechanism
- Pulls data from the sensor's log file, database, or a system port
- Creates the object, declaring it as persistent and inserts it into the database by clustering it into a container, or via methods used by the DOODBMS

2. *Graphical User Interface (GUI)*. Existing GUI components in an analyst's application are also redesigned around the DOODBMS, or new ones are developed. The level of change depends upon how much database code was integrated into the GUI. If the code was already logically separated by functionality, then the change will be minimal. If the GUI components gather information themselves for display rather than using an interface provided by the application, then the changes will be far more substantial.
3. *Communications*. There must exist a means to communicate between the database and its taps and analysts' applications. With a DOODBMS, there may exist a number of components that need to be managed in order for the database to provide concurrency. The communications components needed vary by database vendor. Objectivity/DB requires both a lock server (LS) and connection server (AMS), both of which are described in Chapter II under "Architecture" starting on page 50.
4. *Administrative console*. It is highly recommended that an administrative application be built that allows for the manipulation of the DOODBMS. Unlike Oracle, a centralized management console may not be provided by the DOODBMS vendor. The application should be able to create and delete component databases, build objects that are used and considered common to all the component databases, and modify the existing database distribution scheme's partitions and images.

3.4 Summary

This chapter outlined the procedures necessary for implementing a high-speed IDS data repository designed around a DOODBMS. Borrowing heavily from previous theses involving DOODBMSs and RDBMS to OODBMS schema transformation, I described a number of issues discussed involving the design of the system and how it is to be implemented. These include system design, component design, and testing.

Designing the DOODBMS schema is not much different than designing any other OO system. However, the distributed nature of the system is key. Fragmentation and redundancy of information throughout the domain, discussed in Wu's thesis as "partitioning" and "global schema placement" of objects under high-level design, is crucial to the database's success and will be visited in the next chapter.

IV. Database/System Implementation

4.1 Introduction

The purpose of this chapter is to describe an IDS data repository around a DOO-DBMS. AIDE is used to exercise the methodology of transforming an already existing system designed around an RDBMS into one that utilizes the functionality of a DOO-DBMS. The resulting system is named OOAIDE, for Object-Oriented AIDE.

4.2 Analyze IDS Structure

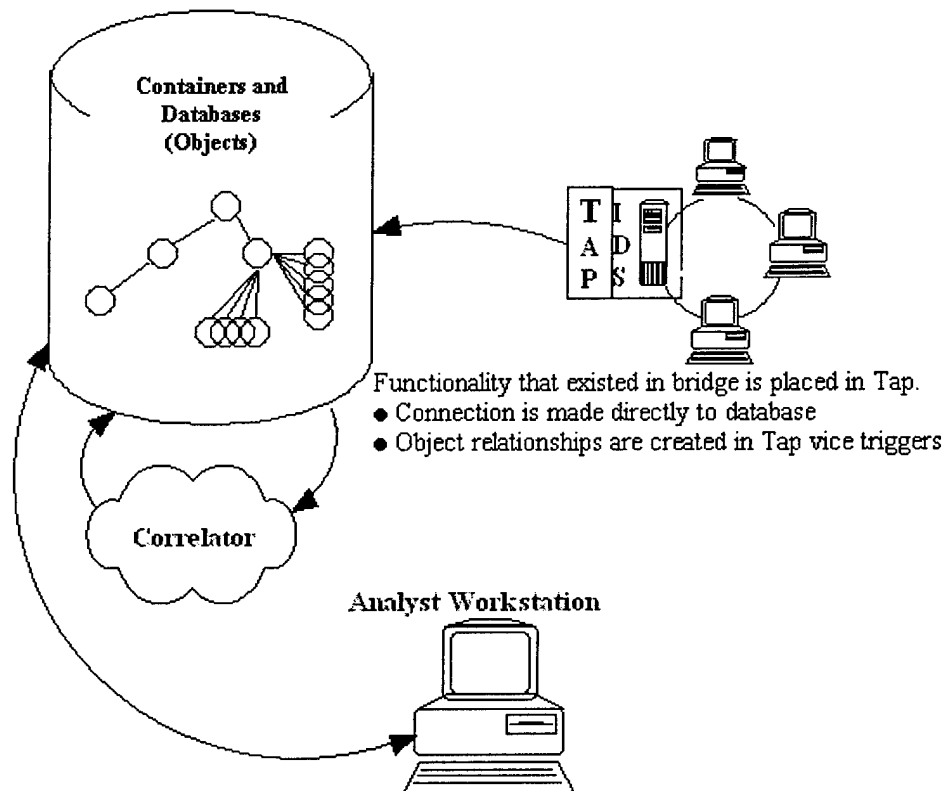
AIDE details are discussed in Chapter II starting on page 16. The following is a brief synopsis of the primary system components and their functions within the system.

- *Taps/Bridges*. These small programs gather information from a sensor located on a host machine and insert it into the AIDE database.
- *Database*. The RDBMS, Oracle, is used to store and maintain the information sent to AIDE from various sensors. Triggers are used within Oracle to perform various functions, such as correlating a sensor signature with that of an AIDE signature and mapping data to the appropriate tables.
- *GUI*. The current AIDE interface, written in Java, is used by analysts to display and manipulate information stored in AIDE. JDBC is used for communication between the application and database.

- *Correlator*. The correlator, also written in Java, runs on the Oracle database server and correlates events together based on AIDE signatures and the time of the event. It is set to query the database for new events at a predetermined time interval, normally one minute.

In the OO scheme, these components change in structure and code based on the change in the database. Figure 4-1 shows the system resulting from the use of Objectivity/DB.

Figure 4-1. OO-AIDE Process Diagram



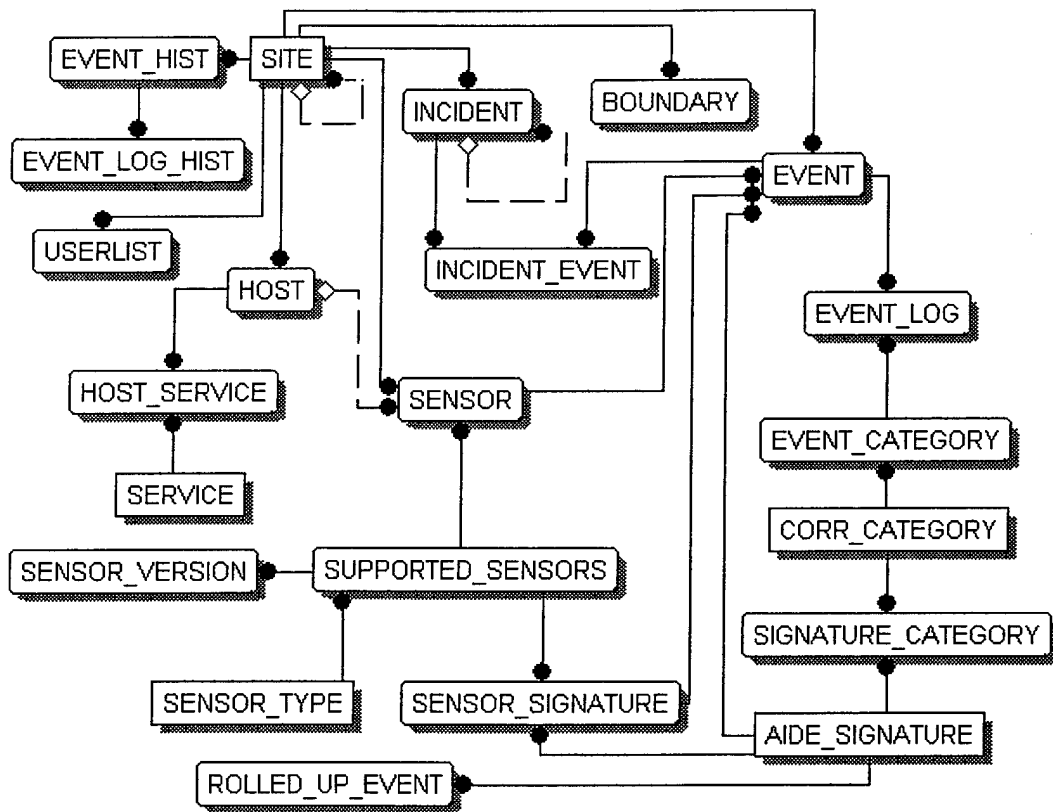
The most significant change from the diagram presented in Figure 2-2 is where functionality is placed to insert information into the database. In the relational approach, a tap communicated with a bridge to insert data. Triggers then processed the data to establish keys and relationships between tables where the data was finally placed. The taps that com-

municate with the DOODBMS do not use bridges, but instead send data directly to the DOODBMS for insertion. Also, relationships are created within the tap and not in the database. This means that each tap takes up a small portion of the overall processing load that once rested on the database server, which allows the database server to service a larger number of taps.

4.3 Re-engineering the RDBMS Schema to a DOODBMS Schema.

The data dictionary is presented in Appendix A and the database itself is described in Section 2.1.3.2. Linhares Lima's proposed steps for converting a relational database to an OO database are applied to AIDE in the remainder of this section.

Figure 4-2. AIDE ER Diagram



4.3.1. ER Model of AIDE

Figure 4-2 presents the ER diagram of the AIDE database. Only entities are shown for simplicity. This model is a direct representation of the functional IDS portion of the AIDE database. Some tables have been left out because they are not yet fully implemented into the existing system or are not within the scope of the research presented in this thesis. No functionality has been lost due to these exceptions.

4.3.2. AIDE Initial Object Model

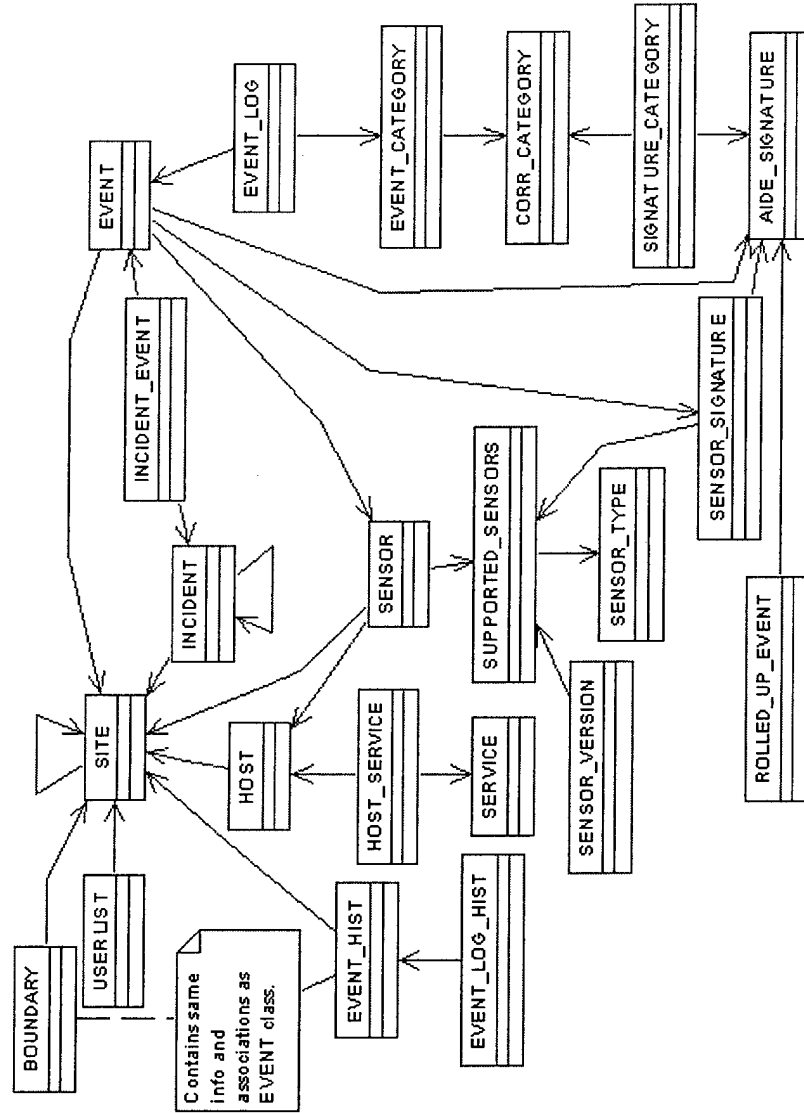
In this step, each entity in the ER diagram is converted to a tentative class. No modifications are made to the class structure. Figure 4-3 shows the results of the translation.

4.3.3. Refine Classes

At this point, the tentative classes are refined to show a more correct representation of the object model versus that of the relational model. For simplicity, a number of Lima's steps are combined to form the overall refined diagram. Specifically, the tentative classes and discovered generalizations and associations within the original diagram have been refined. The result is seen in Figure 4-4.

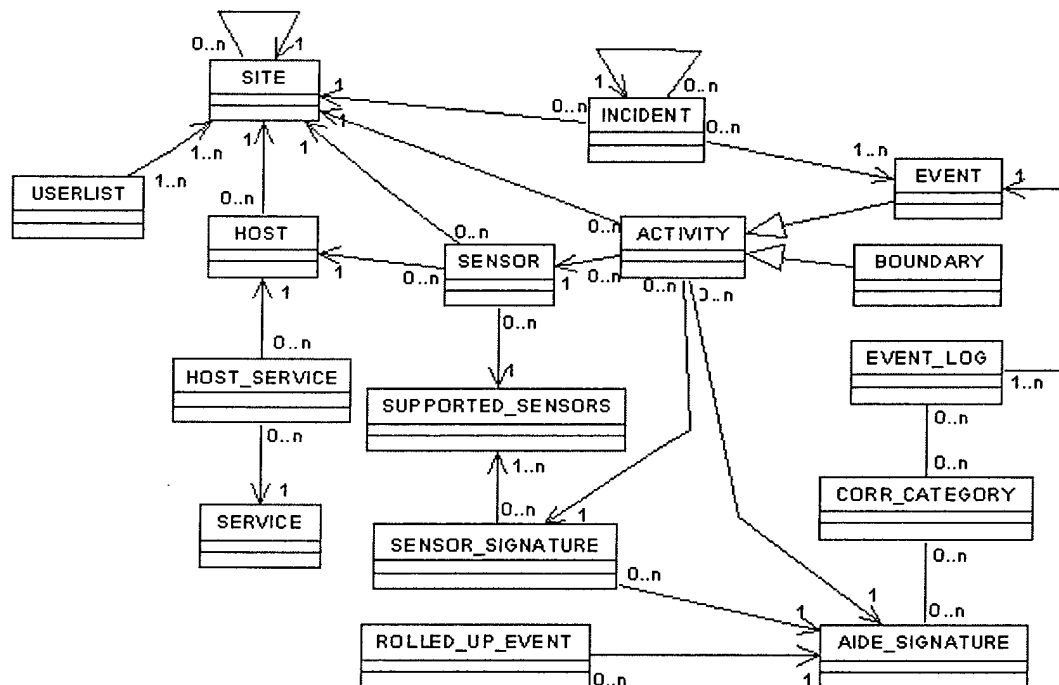
- (a) Combine classes that have same schema. `EVENT_HIST` was combined with `EVENT` and `EVENT_HIST_LOG` was combined with `EVENT_LOG`.
- (b) Discover generalizations. `BOUNDARY` and `EVENT` were found to have similar functionality and attributes. As a result, the superclass `ACTIVITY` was created that generalized `BOUNDARY` and `EVENT`, but maintained any differences between them.

Figure 4-3. AIDE Entity-to-Class Translation



- (c) Eliminate classes (previously tables) that consist entirely of foreign keys to indicate a one-to-one, one-to-many, or many-to-many association (join table). As a result, many-to-many associations are produced between the classes for which the join table was being used. `EVENT_CATEGORY`, `SIGNATURE_CATEGORY`, and `INCIDENT_EVENT` were eliminated.
- (d) Eliminate or combine tables that were used for enumeration or constraints. These tables should not normally be represented as classes. The attributes from the tables that held those values may be rolled in with an associated class that actually used those values. `SENSOR_VERSION` and `SENSOR_TYPE` were rolled into `SUPPORTED_SENSORS`.

Figure 4-4. Refined Class Diagram



(e) Discover associations. In some cases, associations are the result of eliminating classes as in part (d) above. However, in the original AIDE database, tables contained references to rows in other tables by inserting a unique value. In an object diagram, each reference is converted into a uni-directional association. These associations can be seen under the Activity class in Figure 4-4.

4.3.4. Prepare a Functional Model

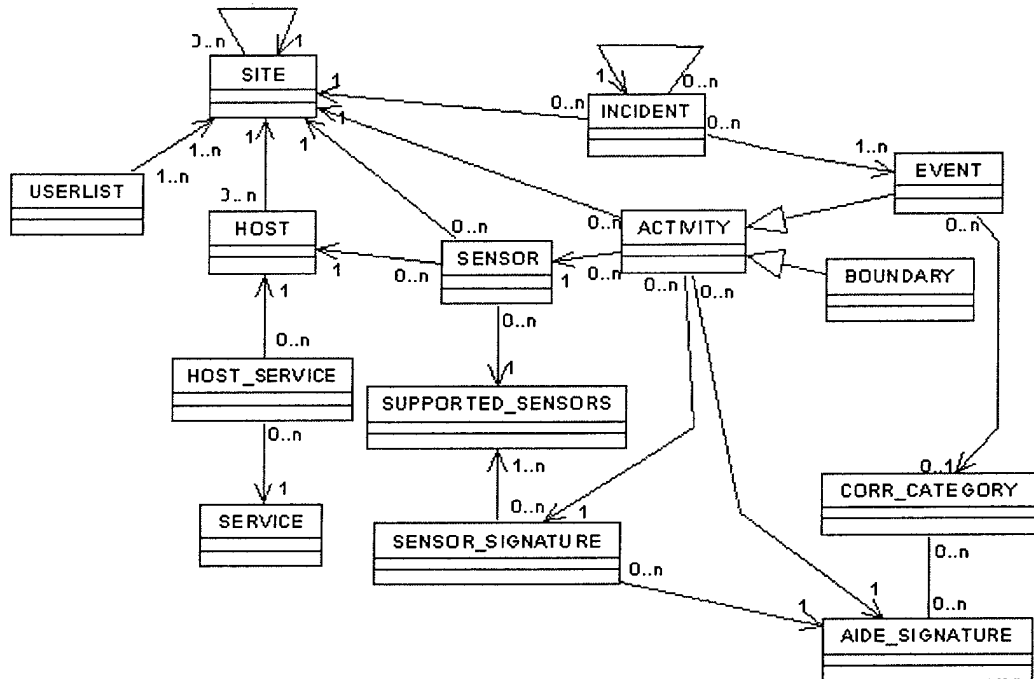
In preparing the functional model, database usage was analyzed against the relevance of the classes that remained in the class diagram. A number of the classes, although allowed to remain in the beginning of the translation process, needed to be eliminated due to the manner in which they were used or the functions that they performed. The following no longer exist in the object database:

- **EVENT_LOG.** This class' only purpose is to maintain a log of events and information such as IDs of messages sent to analysts regarding the event, and the partition that the event was kept in within Oracle. This functionality can be integrated elsewhere within the object model and maintained by the database itself, such as within a specialized container. The elimination of this class is done primarily because the trigger that existed to produce messages and maintain partitions will be eliminated, as will all the other triggers used by the relational database.
- **ROLLED_UP_EVENT.** This class served as a performance enhancement to AIDE. It contained information on each event that is put into the database. The information includes time of insertion, time of last update, AIDE signature, IP addresses of both the event's source and destination, and total times an event with

exactly the same information has been seen. Updating the table for each new event slows the insertion of new events down to a noticeable degree, but queries that involve only the information that this table tracks are much more efficient.

Thus, the resulting diagram is shown in Figure 4-5. The class diagram presented establishes a good baseline and is final with respect to the translation process.

Figure 4-5. Final Class Diagram



4.4 High-Level Design

This section deals with designing the database using the class diagram developed in the previous section.

4.4.1. System Design Trade-offs

Trade-offs defined in Wu's thesis include, among others, performance, memory space, portability, cost, maintainability, and understandability [Wu 3-16]. The following priority has been given to factors important to the design of the IDS data repository:

1. *Performance.* As mentioned in Chapter I, performance is of major concern to the end users of AIDE. Also to be taken into account is the overhead associated with insertion of data and queries made on the database.
2. *Maintainability.* This factor will become increasingly important as the database becomes distributed throughout many sites.
3. *Space.* As the database grows, disk space will be of concern in storing replicated events and site-specific information.
4. *Other.* Portability, cost, understandability, and other factors are also to be considered when the database is being laid out, but do not take precedence over one another at this time. It may be necessary in the future, once an initial design is in place and testing has been achieved, to go back and redesign the database based on the test outcomes and factor in these other considerations as well.

4.4.2. Resolve to-Many Relationships with Respect to Database Representation

Objectivity for C++ and Java contain their own methods to implement many-to-many and one-to-many relationships between classes. The representation of the relationships within the database, however, is language-independent. As a result, interoperability may be achieved between the different languages used to access and modify relationships

in the database. No modifications need to be made to the translated class diagram to resolve any relationships with respect to Objectivity/DB.

4.4.3. Refine Classes and Associations to Better Model Data

Although the database has been translated from a relational model to an object model, it still does not represent an optimized object model to be used as the basis for a DOODBMS. Navigation between the objects is not optimized with regards to performance, which is the primary concern in the new OOAIDE system.

The majority of the class diagram uses uni-directional associations to demonstrate navigation from one class to the next. When considering performance, how the database is to be used may be an indicator as to whether or not a bi-directional association should be used to allow both classes to navigate between themselves. However, it should be used only where needed due to space considerations. A bi-directional association may take up more than twice as much space within the database as a uni-directional association if a many-to-many relationship is needed and if the two classes (or objects) are being kept in separate component databases or containers. This is because OIDs must be stored in both objects and if they are stored within different storage hierarchies, additional data is needed for locating each object. Listed below are some classes where a bi-directional association is required:

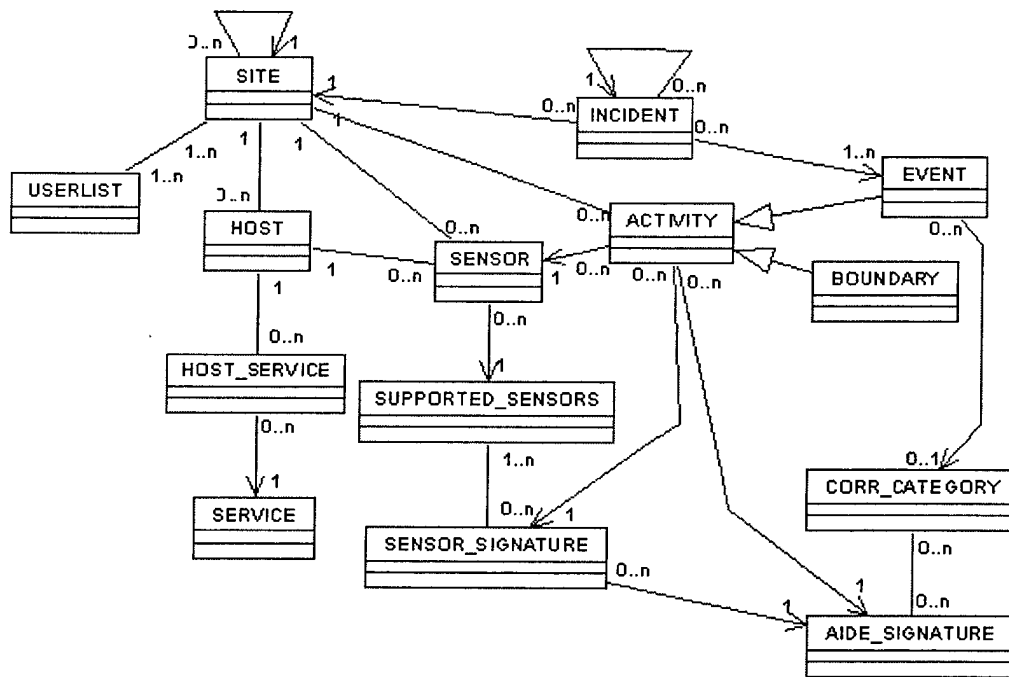
- SITE to HOST
- HOST to SENSOR
- HOST_SERVICE to HOST
- SITE to ACTIVITY
- SENSOR_SIGNATURE to SUPPORTED_SENSORS
- SITE to USERLIST

These classes need access to their parent's and subordinate's information quite frequently. Therefore, a bi-directional association is warranted. There is a problem in providing the Site to Activity bi-directional relationship due to the fact that there are many events being inserted into the database at any one time. For every event being inserted, an object of type Site would need to be updated with the appropriate Activity OID. This performance concern is addressed in section 4.4.6.1.

4.4.4. Design Global Conceptual Schema

The global schema combines the translated database and any changes made up to this point. Therefore, the resulting model takes into account any additional relationships or modifications, as well as their impact on performance, space, and maintainability, but does not yet introduce all the concepts of the distributed design.

Figure 4-6. Global Schema



4.4.5. Distribution Design and Local Schema Design

Probably the most formidable challenge is distributing the database, which includes distribution of objects and object placement onto hard media. The main goals of distributing the AIDE database are to distribute common data so it is kept closer to each site and eliminate the single point-of-failure that exists in the current system. To put it simply, all the needed data should be available to an analyst even if a site, the initial point of collection of data for a sensor located on a given host, goes down. Two different approaches may be taken.

4.4.5.1. Partial Redundancy of Objects

Partial redundancy refers to dividing the objects into two categories: redundant objects and non-redundant objects. Wu identified multiple *data placement* methods by which data may be distributed across a domain [Wu 3-26]. Specifically, there are seven methods by which to distribute objects, ranging from full replication of objects, to placing most of the objects at the most local level to decrease communication. In looking at the needs of the IDS, the following guidelines may be followed to distribute objects:

1. Place objects that will get queried the most often and are the most common to each site in the distributed domain using redundancy. This technique is known as *placement by greatest query rate* [Wu 3-27].
2. Place critical objects at the local sites, but still available to other sites so they may be moved around the domain, or in this case, the federated database. Here, objects are placed near or on the system that is responsible for them, which is known as *placement by system responsibility* [Wu 3-26].

Using the above techniques and reasons, the objects are placed in two distinct categories: redundant and non-redundant objects.

1. *Redundant Distributed Objects (or classes)*. These objects are redundant, as stated above, so that they may be updated easily across the federation and so that they may be located quickly by each site to reduce network communication.

To achieve redundancy under Objectivity/DB, an image is located within an autonomous partition in each database that is to contain replicated data using the Data Replication Option (DRO) and Fault Tolerant Option (FTO). The following types of objects are deemed redundant:

- (a) **SITE**. Site objects are needed by every analyst workstation in order for those analysts to traverse events located at each site. Therefore, the objects are stored at each site.
- (b) **SUPPORTED_SENSOR**. A listing of supported sensors is available at each site for the system and analysts.
- (c) **SENSOR_SIGNATURE**. A listing of signatures for each sensor type is available at each site for each system and analyst.
- (d) **AIDE_SIGNATURE**. A listing of AIDE specific signatures is available at each site for each system and analyst.
- (e) **CORR_CATEGORY**. A listing of correlation categories for the AIDE signatures is available at each site for each system and analyst.

(f) SERVICE. A listing of common services that a host may use, such as FTP and HTTP information, is available at each site for each system and analyst.

(g) USERLIST. A listing of all the users in the system and what sites they are able to access is also made available to each site.

2. *Non-Redundant Local Objects (or classes)*. The following objects are specific only to the site upon which they were instantiated and stored.

Fault tolerance is achieved through the FTO in Objectivity/DB. This option used alone, without DRO, allows data to be placed in the federation and made available even if the rest of the federation goes down, but not replicated throughout. The following types of objects are made local:

(a) HOST

(b) HOST_SERVICE

(c) SENSOR

(d) ACTIVITY. Includes BOUNDARY and EVENT through inheritance.

(e) INCIDENT

4.4.5.2. *Full Replication of Objects*

The above method enhances performance in the context of insertion of data, since a quorum of images does not need to exist to write out to a redundant distributed database, but fails to address single points-of-failure for the most critical data: events. A dis-

tributed architecture is needed that allows the taps to write out events to a *main* database image, allow those events to be replicated so that if a site goes down the data can still be accessed elsewhere, and still allow common site and sensor data to be co-located and manipulated so as to not interfere with events.

A denial-of-service (DOS) attack on a network is one example. If a site machine residing on the network becomes a target of the attack, or if it becomes overly saturated with input from its subordinate taps, the data that resides on the machine could become unavailable. Data includes any events recorded from a host's taps regardless of whether or not they occurred during or before the DOS. As a result, the analyst may not be able to build a complete picture of what happened with the site at any given time that the host has been collecting data and sending it to a site.

The events themselves can be made redundant so that they are stored throughout the federation at every site containing an image of the database. The drawback is that redundancy does take up a great deal more disk space and more processing capability at each site. This balance of availability, performance, and storage capacity will have to be weighed by an administrator in determining how to partition the database. A possible solution is presented in the next section.

4.4.6. Implementing Using Objectivity/DB

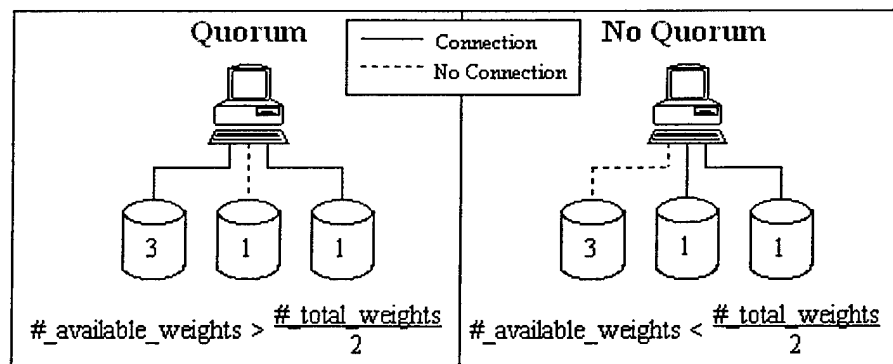
As explained in "Distribution" starting on page 56, autonomous partitions and images are used to separate distributed data from localized data. At this point, all of the classes in the previous section are located in partitions for reliability (fault tolerance) reasons. However, partitions must also contain images of databases to be made redundant. Objectivity stipulates that an autonomous partition can *control* only one image for a redundant database, but may contain many number of database images. A partition may also contain multiple databases that are not redundant.

This thesis presents an implementation consisting of full redundancy for the sake of *reliability* and *availability* of data to analysts. These two factors are important considerations in deciding whether or not to maintain copies of data at every site [Blahe 411]. Although this can create problems concerning performance, the problems can be divided so as to distribute any negative effects on the system.

4.4.6.1. Delegate Data Distribution Using Unweighted Quorum

A quorum exists when a majority of the copies of a replicated database are available for writing. A quorum does not need to exist to read from the database, although such reads may result in reading stale data. The quorum is based on adding the values of the weights of the different images and deciding if the resulting value is a majority of all the weights combined. Thus, it is possible to assign weights to images so that a write will occur even if only one or a few of the images are available. Weights may be changed at any time to reflect changes in the distributed architecture. Figure 4-7 shows an example of a quorum and a non-quorum based on images with unevenly distributed weights.

Figure 4-7. Quorum and Non-Quorum with Unevenly Distributed Weights



It is possible for the number of available weights to equal the number of unavailable weights. In such a case, a tie-breaker partition with a weight of one is used to produce a quorum. The tie-breaker partition contains no databases and is called only when a tie is to be broken between the number of available and unavailable weights.

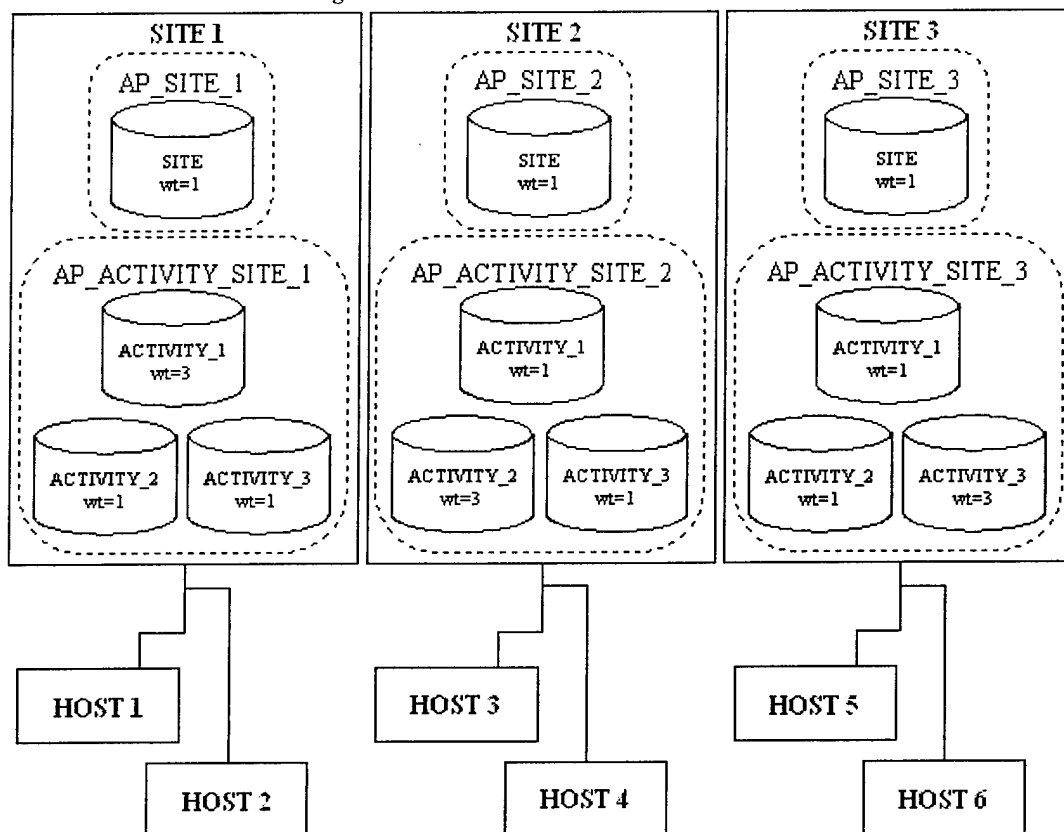
The classes are split up once more. In this iteration, the concentration is not on the redundancy issue, since it was decided that all objects will be made redundant. Instead, what is needed at this point is a way to delegate where the objects are to be placed, and more specifically, what databases they will be placed in.

- **SITE Database.** This database contains all of the objects of the type of classes listed in the previous section under *Redundant Distributed Objects*.
- **ACTIVITY Databases.** These databases contain all the activities, including the objects of type **EVENT**, **BOUNDARY** and **INCIDENT** for each site. Additionally, they contain objects of the type of classes specific to the site on which the database is placed to include **HOST**, **HOST_SERVICE** and **SENSOR**.

The reason for splitting the objects up in such a manner is primarily a matter of performance and availability. Since Objectivity will be updating the event objects quite frequently, it is desirable to place those objects away from the objects that are updated less frequently to avoid concurrency issues involving the locking of a container or database. Thus, the two databases are used as logical separators between object types.

Next, these databases are placed such that availability and fault tolerance are at a maximum. For this, multiple autonomous partitions (AP) are used to maintain the different databases according to redundancy needs. The **SITE** database and its images are placed in one AP at each site and the **ACTIVITY** database and its images are placed in another AP. Figure 4-8 below shows a graphical representation:

Figure 4-8. OOAIDE Databases and Partitions



For the SITE database, the weight is set to one (1) wherever an image is placed. A SITE, HOST, SENSOR, or any other object of the type that is contained within the SITE and database *cannot* be added unless a quorum of the images are available. This ensures that every site contains a proper image of sites used throughout the system.

However, the ACTIVITY databases are quite different. The weights are set such that data will be written so long as the site which controls the database is available. The data is still replicated throughout the OOAIDE system, but priority is given to a host's site so that in the situation where a majority of sites are not available for writing, data can still be entered at the local site. A simple calculation can be used to calculate the weight distribution:

weight of image at controlling site = # of database images of the database

weight of image at any other site = 1

Figure 4-7 and Figure 4-8 both use the calculation to assign weights to the different images. In Figure 4-8, if ACTIVITY_1 is unavailable, then neither HOST_1 nor HOST_2 can write to the ACTIVITY_1 database. However, any other site can still read from their image of ACTIVITY_1, but writes are not allowed. If the opposite is true and all the remote images of ACTIVITY_1 are unavailable, but SITE_1's image of the database is available, then any site or host can read and write to the ACTIVITY_1 database that is located at SITE_1.

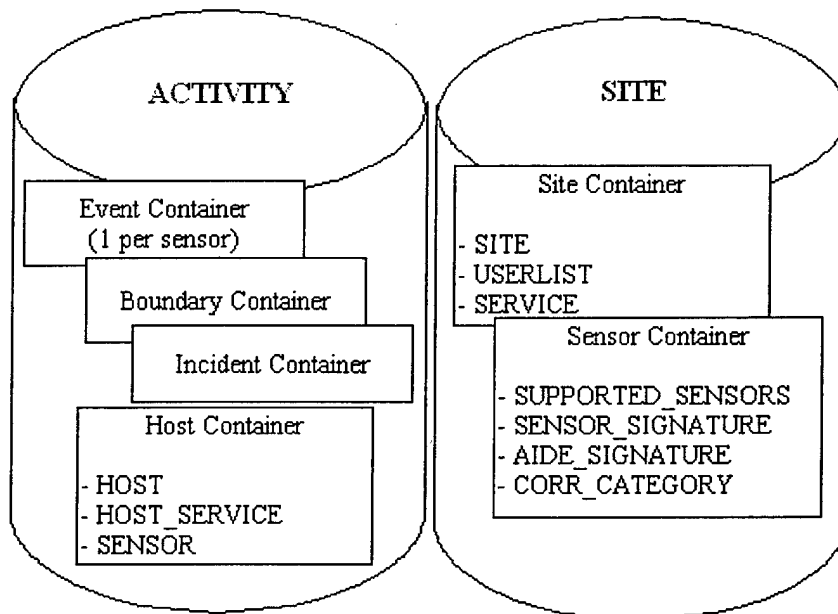
Even with the federated database split up into multiple databases, there still exists a problem with all of the sensors inputting information into a single database. At this point, there needs to be an even further breakdown in the storage hierarchy. This is where *containers* are used. Objectivity/DB provides the following guidance to ensure concurrency [Objectivity/Java 99]:

- Assign all components of a composite object to the same container if the entire composite object will be accessed as a unit.
- If a composite object is large and complex and can be divided logically into subsystems that may be modified independently, store the objects that make up each subsystem in a separate container.
- If a large number of objects are read frequently but rarely updated, you can safely assign them all to the same container.
- Distribute objects that require frequent update among as many containers as reasonably possible.
- Keep shared resources in separate containers from objects that use those resources.

- Use multiple readers, one writer (MROW) sessions to help manage applications that require containers to be locked for long periods of time.

As mentioned under “Architecture” starting on page 50, locking of objects occurs at the container level. With many events expected from multiple sensors located throughout the network, each sensor writes to its own Event container. This allows each sensor a nearly uninterrupted flow of information. It is understood that GUIs and other analysis tools will need to connect and query information from the Event container, but with MROW it can be done without interrupting the sensor’s input. Figure 4-9 shows a detailed diagram of the container breakdown.

Figure 4-9. OOAIDE Database Containers



The Boundary container is another potential bottleneck within the system. The input to the boundary container from routers and firewalls that produce information pertaining to those addresses that the system has rejected from entering the network without the IDS even seeing the traffic may be more than the input into the event container from the

taps. These systems are also allocated their own containers due to the high volume of entries.

4.4.6.2. *Limitations of Design*

The distributed design presented above has some limitations.

1. **SITE updates.** For SITE images to be updated, they must participate in a quorum at a given point in time. This means that the process updating the SITE database must be able to physically communicate with a majority of the sites to establish a quorum to get some of them updated. At a later time, the other unsynchronized sites must participate in a quorum to get updated or they may contain stale data.
2. **ACTIVITY updates.** The same limitation applies to the ACTIVITY images held at their respective sites. Communication must be available between a quorum of sites if the data on any given controlling site is to be replicated. This, as far as AIDE is concerned, could be a major problem if multiple organizations are involved and they do not wish to leave network ports open for communication outside of their organization. Possible lack of communication is the primary reason why image weights were distributed as they were in Figure 4-8.
3. **Decrease in performance.** As more hosts are distributed throughout the IDS domain, a decrease in performance can be expected due to the added stress to the sites that contain images of the ACTIVITY database and the communication needed to establish quorums between sites.

Regardless of the above limitations and stipulations concerning communication amongst the nodes, the primary images are still availability and distribution of common data is achieved. It should be noted that the above scheme, which purposely parallels the current AIDE scheme, may easily be broken down to allow for another layer within the hierarchy so the databases could be further distributed with respect to each site.

4.5 Low-Level Design

This portion of the thesis describes the high-level design and implementation using lower-level design structures [Wu 3-28 - 3-33]. The only portions of the IDS built at this point are the data management components represented in Figure 4-5.

1. Determine object representation. This step involves implementing the data within the database objects, or attributes, as either primitive types or as other referenced objects. In the end, however, all attributes should be maintained as primitive types.
2. Refine inheritance structure. After the database translation, there is no need for any additional inheritance. All like-classes have been either integrated or modified using inheritance.
3. Implement object methods. Methods are assigned based on responsibility, and new classes may be added later during coding. As a minimum, *set* and *get* methods for each attribute made available for reading and writing are to be added. Additionally, needed operations also include *set* and *get* methods to establish or return any primitive type necessary within the application and overloaded constructors.

4. Establish object visibility. For the IDS data repository used in this thesis, visibility is established through relationships between objects. There are other ways of obtaining the needed visibility, such as, inheritance, making one object an attribute of another, implementing one object as a static class, and passing the needed object to all the objects that will need to use it.
5. Identify polymorphic methods. In an OO program, methods may need to be implemented lower than the superclass in an inheritance hierarchy. In the current implementation, there is no need.
6. Apply OOD using OOPL. Objectivity for Java is used to build the database. Objectivity/C++ is another alternative, but the programs and taps are primarily for testing and demonstration purposes and Java provides an ideal rapid prototype environment. In the future, C++ bindings can be used to incorporate more performance into the system and create a smaller footprint on the tapped machine.
7. Perform physical design. Here, the database is built using Objectivity/DB and the programming language of choice. The design, ready to be implemented, is laid out on disk and files are established to hold data. Using Objectivity/DB, the federated database is established via command line input. The databases, containers and objects are all built within Java. The images to the various databases are also established via command line input.
8. Design user interface. Java is also used to design the user interface. This is not a major focus of this research, so only a rudimentary program will be used to show data in the database.

4.6 Tap Application Development

To develop the application to insert data, a general well-known OO methodology is used. In his thesis, Wu describes only in limited detail the process of designing the application to access the database. Therefore, the Object Modeling Technique (OMT) will be used to transform the standing notion of what a tap should accomplish into an object model that can be translated into code [Blaaha 118-119].

4.6.1. Conceptualization

The tap application is used to retrieve information from a sensor's data store for insertion into Objectivity/DB. Due to the heterogeneous nature of the networks upon which the taps will exist, they are built as cross-platform applications so a great deal of code reuse is realized.

The systems where the taps will reside may also be heavily loaded due to the systems' monitoring of high volumes of network traffic. Therefore, the applications are as small as possible and consume only the resources that are absolutely necessary to accomplish the needed tasks.

A diagram depicting the OOAIDE system is the same as that in Figure 2-3 for the existing AIDE system. However, instead of the tap communicating with Oracle, it communicates with Objectivity/DB through a set of configurable ports used by the DOODBMS for distribution and object locking.

4.6.2. Analysis

The following capabilities are present within the tap application. This list was gathered using analysis of the existing system and knowledge gained while interviewing a number of its technicians.

1. Since the sensors exist on a variety of platforms, the tap is able to conform to those platforms.
2. The tap is able to read from files, relational databases, object-oriented databases, and data streams where sensor data may be stored.
3. The tap is able to insert into Objectivity/DB all events/boundaries that are processed by the tap application.
4. The application is written such that as few changes as possible are needed to allow it to interface with another type of data storage device or hardware platform for retrieving information.
5. Since the system is distributed, network connectivity is imperative.

In order to model the above behavior, a set of classes are generated. The process is similar to the process used in the beginning of this chapter for refining a class diagram once the tables were converted to tentative classes. The main difference is that the analysis starts with a set of capabilities that the application should provide versus a detailed ER diagram. The following classes are used to model the behavior:

1. *OOAIDE*. Top-level application class that “ties” the system together. Because multiple types of taps or other classes could access the DOODBMS in the future, this class serves as a level of abstraction between the remainder of the application classes and the data that they will access. The level of abstraction is also needed in case of future changes made to the database. Instead of updating each class when

objects are moved around to various containers or databases, this class will be called so it can place the data in the correct place within the federation. It provides functions to do the following:

- Connect to the DOODBMS
- Open and close various partitions, databases, and containers for access by other classes
- Retrieve a session and policy for other applications to use in accessing persistent data
- Provide a set of *finder* methods to be used in scanning the databases and containers for specific objects that conform to a query
- Provide a set of accessor (*get* and *set*) methods for applications to add objects to the database that only exist at the container level, like those objects of class `SITE` and `SUPPORTEDSENSOR` that do not have a higher-level containing entity
- Perform high-level administrative functions for building, deleting and reinitializing containers and databases

2. *Tap*. The *tap* class serves as a superclass to all taps. There are a variety of common functions that each tap needs when connecting to and using the DOODBMS. Other than maintaining references to the various storage objects, such as a partition

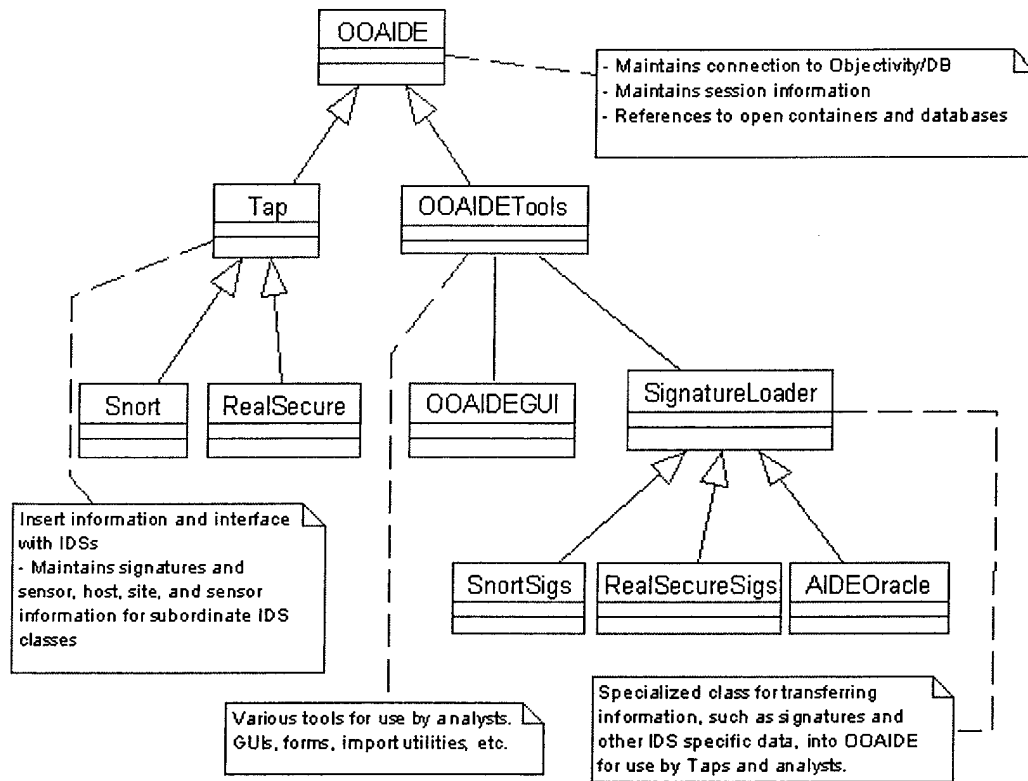
or database, the tap needs only session information to insert events. The methods in the OOAIDE class provide the functionality needed to place the information into the database. The functionality includes:

- Maintain references to the tap's site, host, sensor, supported sensor, and sensor signatures
- Provide methods for using other databases, files, ports, or other storage media where from sensor information is to be retrieved
- Create events for placement into the DOODBMS
- Create relationships between events and other objects as designated by the OOAIDE object diagram (DOODBMS schema)
- Inherit from OOAIDE in order to provide overall database functionality to the various taps

3. *Tap subclasses.* Inheriting from Tap, these subclasses represent the various taps needed by the overall OOAIDE system. They are the heart of information retrieval and provide the various methods needed for the Tap class to be able to function in a variety of environments.

4. *OOAIDETools.* The tools class extends OOAIDE and provides the architecture to support various tools for use by an administrator or analyst. In Figure 4-10, the utilities for loading signatures for various IDSs and a GUI are associated with the OOAIDETools class.

Figure 4-10. OOAIDE Class Diagram



4.6.3. System Design with OODBMS and Detailed Design

This step is where coding and integration takes place. In the final application design, there are a number of classes that are not navigable from any other class; SITE and SUPPORTEDSENSOR are good examples. For this reason, directly accessing the containers that contain these types of objects is necessary. Instead of integrating this functionality throughout the code, which can be very costly in terms of maintenance, the top-level class, OOAIDE, maintains the container references.

4.7 Summary

This section described the implementation of a working model for a complete object-oriented system to maintain data from multiple IDSs. The process involves converting the AIDE database into an object form and then building the application specific to storing persistent objects in the new database. The evaluation of the proposed design is presented in the next chapter, along with a comparison of the new design to the existing AIDE system.

V. Testing and Evaluation

Three enhancements that a DOODBMS would provide over an RDBMS are given in Chapter I under “Research Hypothesis” starting on page 6. In keeping with the hypothesis, there are three main focal points for testing and evaluating the design:

- *Performance Benchmark.* Measure throughput performance of inserting data gathered from a sensor tap into both Objectivity/DB and Oracle 8i.
- *Demonstrate distribution.* Demonstrate the proposed distributed design of the database as given in “Implementing Using Objectivity/DB” starting on page 82.
- *OOP Language Standardization.* Demonstrate that multiple taps may be built from the same object-oriented design and OOP language to provide consistency within the overall architecture.

5.1 Performance Benchmark

For performance testing of the proposed design, this thesis follows a process used by Tomaz Domanjko, a researcher at the University of Maribor in Slovenia. The procedure was published in *Java Report* for evaluating OODBMSs against RDBMSs for object persistence [Domanjko].

This section will focus on two metrics: record/object insertion and query throughput given increasing numbers of objects, and the amount of programming code required to accomplish the task. The latter, as stated by Professor Domanjko, is one of the most popular software metrics.

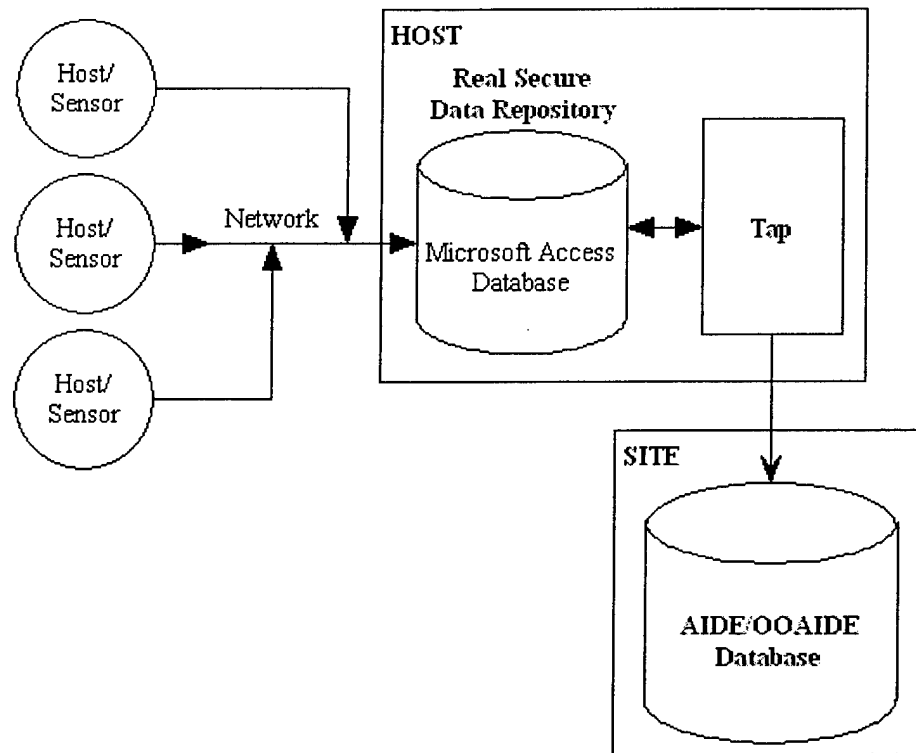
5.1.1. Testing Steps and Architectures

In order to gauge the difference in performance between the two databases, test results from insertions and queries against AIDE and OOAIDE are compared.

5.1.1.1. Insertion.

The test case for the insertion includes taps for the Real Secure IDS from ISS. The architecture of this tap is given below.

Figure 5-1. Real Secure Tap Architecture



The current AIDE tap for Real Secure was built using Perl with a number of libraries to enable access to Oracle and other relational databases. The Perl program code and needed Oracle PL/SQL triggers are listed in Appendix C. The programming model used for this tap is not common amongst the taps used within AIDE. Most use a tap/bridge architecture programmed in C, but this tap connects directly to the database and does not require

the extra overhead of transmitting data to a bridge and having the bridge insert the data into the database through the triggers. The following functionality is implemented into the new OOAIDE tap:

- Connection to Real Secure Microsoft Access database
- Connection to OOAIDE database
- Retrieval of data from Real Secure database. In Java, this will be done using the generic Java Database Connectivity API (JDBC) [Farley 190]
- Begin timer (only for testing)
- Iterate over data and insert it into Objectivity/DB
- End Timer (only for testing)
- Have the process “sleep” for a predetermined amount of time before retrieving additional records. The sleep time for the current AIDE tap is one minute

There is some additional functionality is not implemented due to its irrelevance to the test environment. This includes:

- Real Secure database synchronization procedure. This code is responsible for synchronizing the database with the network sensors so that any data residing in the Real Secure main program’s cache, which resides on the host database machine, is written to Access. The database used in this benchmark is static.

- Routine for choosing which Real Secure sensors are to be synchronized with the database. Because the OOAIDE tap is not synchronizing the engine and sensors, there is no need to specify these parameters.
- Heartbeat signal. Ordinarily, the AIDE tap sends a heartbeat signal, which returns whether or not the Oracle database is available for connection. The test OOAIDE tap assumes that the OOAIDE database is online and available.
- Trigger code includes checking of the BOUNDARY table to see if an event was caught by a firewall or router. This functionality is not included due to the fact that the AIDE system was not fully utilizing this table at the time of this writing. Additionally, there are alternative ways for OOAIDE to provide this functionality without comparisons to objects of type Boundary stored in the database at the time that Event objects are instantiated.
- Trigger code also includes a ROLLED_UP_EVENT lookup and update procedure necessary for updating the EVENT table. This functionality is not included. The ROLLED_UP_EVENT table is implemented to simplify the retrieval of specific event information and mainly maintains counts of how many events have occurred for a given signature. There is no equivalent in the OOAIDE model because such constructs are better implemented at the container level rather than as objects.

The test case for insertion of records involves the following:

- A standardized test case for insertion must be built that demonstrates common functionality across taps using both databases. To ensure that taps are built in the same manner, a current sensor tap used with Oracle 8i and the trigger code that the tap utilizes to insert its data is reverse-engineered. Another tap is then built using the OO architecture proposed in “Tap Application Development” starting on page 91 that provides the same functionality and stores the same data.
- Data is collected from a common data source located on the host machine.
- Testing is limited to the insertion of records into the target database. As a result, functions that read data from the sensor and process it on the host machine prior to being inserted are not reflected in the results. Preprocessing of the information, however, does not include fetching host and site data from the target database to link records or objects together to establish relationships. This is a necessary function in inserting the records/objects and is included in the results.
- A number of records are inserted into the database. Each record corresponds to a single event and all records are of the same size. The test is run with 100, 500, 1000, 2000, 5000, and 10000 records. Each set is run ten times to establish an overall average.
- Identical machines are used to run the tap and database over the same network.

5.1.1.2. Queries.

Four queries are used to measure performance at various levels within the object hierarchy. The four queries range from retrieving a large amount of records, as

would be the case for an analyst's tool who might be correlating a large number of records, to a small number of records, such as pinpointing the search space to a specific known hostile IP address. For simplicity, the query descriptions are given in the next section dealing with the results of the tests.

All queries are built using Java 1.3 utilizing JDBC and Objectivity's own connectivity method, AMS. Furthermore, they are compiled using Sun's JDK1.3 javac compiler with the '-O' switch indicating that the classes should be optimized for runtime efficiency. All four queries are reproduced a total of ten times each for each database, and then each set of runs are averaged to get the overall result for the particular query. The data used in the queries is the same data inserted by the Real Secure taps in the previous insertion test. Before the queries are made, the databases are checked to make sure the data each holds is identical to the other. As the results are returned, the number of objects/rows are also checked to make sure the databases are each returning the correct results.

5.1.1.3. Test Data

Test data to insert into the two databases originates from the Microsoft Access database used by Real Secure. The events input into the database are the result of using Nessus, a remote security scanner, to probe several machines on the network where Real Secure is running [Nessus]. The scanner is run numerous times to allow the number of events in the database to accumulate. Each event corresponds to a single record in the database. The Real Secure database, specifically the RSLog table, contains the information in Table 5-1. The fields in bold print are retrieved and inserted into AIDE and OOAIDE:

Table 5-1. Real Secure Database Fields

Event Date	ICMP Type
Event Name	ICMP Code
Protocol	Event Priority
Source Port	Kill Action Specified
Destination Port	Source Ethernet Address
Source Port Name	Destination Ethernet Address
Destination Port Name	Raw Data Length
Source Address	Raw Data
Destination Address	Decode Pair Count
Source Address Name	Engine IP
Destination Address Name	Pulled to Enterprise Database (boolean)
TCP Flags	Engine Type (Network or Host)

The Real Secure database consists of events from the following types of attacks: SYN Flood, HTTP Shells, Windows Access Error, FTP Bounce, FTP Privileged Bounce, FTP Syst, Email Ehlo, Portscan, Pingflood, Backorifice, Email turn, HP Open-View SNMP Packdoor, HTTP Java, Mstream Zombie, Queso Scan, SNMP Suspicious Get, SSH, Sun SNMP Backdoor, and IP Halfscan.

5.1.2. OOAIDE Real Secure Test Results

Two series of tests, insertion and query, are conducted on the two systems and the results are compared. The results of both tests are shown in Figure 5-2 and Figure 5-3. Table 5-2 shows the hardware configuration used in all the tests. The values given for the insertion experiments are averages of five iterations for each of the object counts indicated on the x-axis. There was almost no variance in the samples gathered for the insertions. All of the reported times are within one second of the times indicated on the chart for each of the tests.

The query experiments were run ten times for each query indicated on the x-axis in Figure 5-3. The variance surrounding the average for each query is indicated with an error bar on the chart.

Table 5-2. Hardware/Software Test Configuration

Machine	Role
933 MHz Pentium III 256 MB RAM 7200 RPM MAXTOR HD	OOAIDE & AIDE Database Server MS Windows 2000 Oracle 8i Objectivity 5.2
400 MHz AMD K6-3 256 MB RAM 7200 RPM IBM HD	Host platform executing Real Secure taps MS Windows 2000 ActiveState Perl 5.6 Personal Oracle8 Client Objectivity 5.2 JDK 1.3
233 MHz AMD K6-2 64 MB RAM 5400 RPM MAXTOR HD	OOAIDE Replicated Database RedHat Linux 6.2 Objectivity 5.2

Figure 5-2. Object Instantiation and Record Insertions for AIDE and OOAIDE Tap

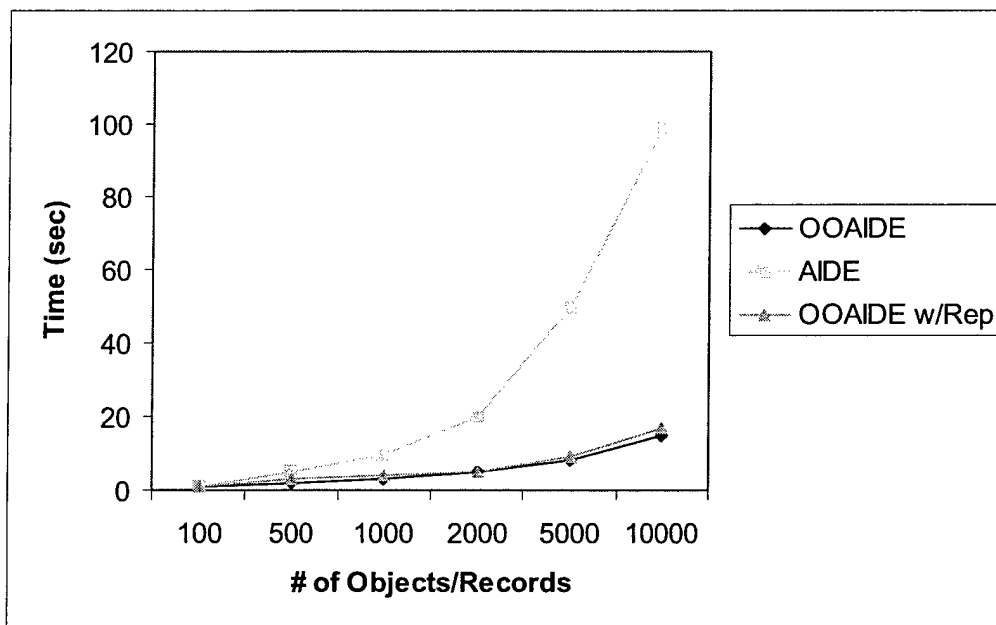
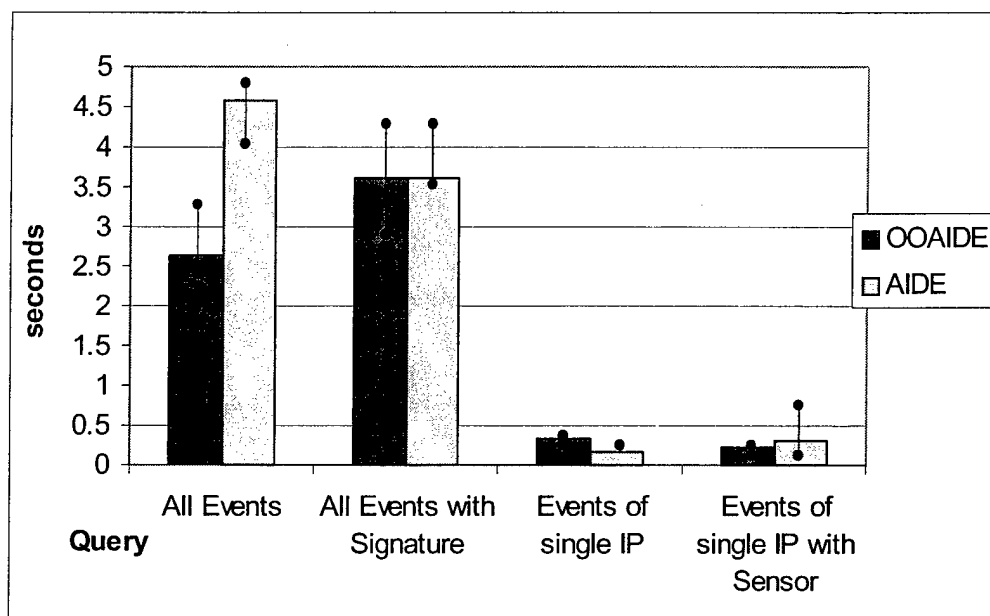


Figure 5-3. Queries on AIDE and OOAIDE via Java



5.1.2.1. Insertions.

The results given above would indicate that OOAIDE is at least 2-3 times faster when there are only several hundred objects/records instantiated or inserted into each of the two databases. However, those numbers increase dramatically around 2000-10000 objects/records.

There are two primary reasons that contribute to OOAIDE's insertion performance increase over AIDE.

1. No table lookups. Since OOAIDE resembles a pure object model, there are no table lookups to gather information needed to maintain relationships between objects. In AIDE, there are two tables used to gather a number of pieces of information that are needed to insert an event into the EVENT table. When a record is first submitted to the database for insertion, a trigger calls the

SENSOR_SIGNATURE and AIDE_SIGNATURE tables to get the priority, category, and signature name.

In the OOAIDE model, there are relationships that still need to be maintained, but resolving references amongst the objects is occurring much faster. First, the relationships are being maintained on the host side and not the server side. This means that while the workload would increase for the AIDE database, the work is distributed to the hosts in the OOAIDE database. Likewise, through the use of hash tables, hash maps, trees, or any other Java construct that allows for storage and fast retrieval of objects, relationships can be built significantly faster than table lookups will allow since only a subset of data is being queried. For example, it would be unwise to perform a query on the DOODBMS for a sensor's signature every time we need to build an object that requires it. Instead, a hash map can be built that stores the objects only pertaining to that particular sensor and every time one is needed, it is in memory because a reference to the object is being maintained by Java. This is known as anticipatory client-side caching [Loomis 194]. To keep the references synchronized with the DOODBMS, a simple iteration through the construct can be performed and the objects will be updated automatically.

2. Another factor apparent in testing the databases is that inserting large numbers of objects at a time scales better in the DOODBMS. In the AIDE tap, clusters of 50 records are sent to the database for insertion at any one time, which means that as the number of objects to be inserted increases, more transactions are needed to send them to the relational database. However, the OOAIDE tap doesn't need to commit objects until the transaction is to take place for all of the objects. The

objects are made persistent when instantiated and manipulated in the host's memory until a commit method is explicitly called. This means that network transmission and the work needed in communicating with the database is used efficiently.

5.1.2.2. *Queries.*

Queries against the two databases provided mixed results, none of which were conclusive. There are a number of reasons, however, for the results that were gathered for each query.

The Oracle database would've been much quicker for a number of the queries had the ROLLED_UP_EVENT table been used. However, the tests were written such that traversals through the object hierarchy were necessary to retrieve information from multiple RDBMS tables and multiple DOODBMS containers without any optimizations. Each of the four queries are explained below:

1. *All Events.* The first query involves the retrieval of all the events in the database - 10393 records/objects. There are no relationships involved and as the results are received, objects of type Event are instantiated. The difference in time, as indicated in Figure 5-3, can most likely be attributed to the overhead within the Oracle database and efficient paging of the Objectivity/DB. Oracle must first retrieve the records from its table and then send the specified columns and rows over to the application through JDBC. In contrast, Objectivity/DB has stored all the events in a single container where they are laid out on disk in successive pages. When event objects are requested, it begins sending pages to the application via AMS. It is up

to the application to filter the pages for the needed objects. In this case, there is no overhead associated with filtering the pages because all objects are to be included in the results.

2. *All Events with AIDE Signature.* This query involves the selective retrieval of events that contain an AIDE signature reference or column value not equal to "null." In other words, there was a valid AIDE signature for each event regardless of whether or not a sensor signature is present.

The results of this query show that both databases perform equally well. Because only certain events are extracted from the databases, Oracle is just as efficient as Objectivity/DB. Oracle, by design, is expected to perform such queries very efficiently. The main reason why Objectivity/DB does not perform as well on this query is due to the fact that reference fields, such as those looking for AIDE signatures, cannot be used in a query. Therefore, all the records must be retrieved from OOAIDE and a separate Java routine run on the objects to determine if the AIDE signature relationship actually associates the given event to an AIDE signature. If the relationship reference is "null," then the event is dropped from the selection. Objectivity/DB performs the queries as well as the Oracle database even though the application doing the majority of the work is on a machine half as fast as the Oracle database machine and a separate Java function is run after all the events have been retrieved.

3. *Events of a single IP.* This query involves retrieving events given a single IP address. Out of the 10393 records/objects, 7256 are retrieved. As mentioned above, Oracle excels at simple, single-table queries. In this case, however, OOAIDE is slightly quicker than AIDE. This is mainly due to the fact that the IP address is indexed within the OOAIDE database. Looking at queries from the current AIDE graphical user interface (GUI), it is determined that the IP address of the source of an attack is of major interest.

4. *Events of a single IP with Sensor.* In this query, records/objects of type Event are retrieved from the databases with the intention of also capturing information concerning the Sensor that placed the event in the database. In contrast to the previous query, OOAIDE performs slightly better than AIDE in this test case. The reason stems from the fact the AIDE must first join the Sensor and Event tables to gather the appropriate information. In OOAIDE's case, the Sensor relationship is passed to Java along with the Event. There is no need for the a join to take place in OOAIDE and the reference is simply retrieved through a normal 'get' method call.

5.1.2.3. *Code Requirements for AIDE vs OOAIDE*

Code requirements for each test case, insertion and query, differ given the task. In the case of insertion, the amount of code does not differ dramatically at first glance. However, further analysis shows that the code required by AIDE to insert events is kept in three different places within the system and calls between those portions of the system take additional time and resources. The insertion code for AIDE and OOAIDE is given in Appendix C.

For queries, the difference in the amount of code required to accomplish the task is more dramatic. The actual code for each query is listed in Appendix D. SQL, by nature, does not fit well into the object-oriented programming model. As a result, Java, and other languages, must provide wrappers or specialized methods to integrate SQL into the environment. In Java, Statements and Resultsets are used to gather data returned from a JDBC connection SQL query. These Resultset must then be iterated over and the individual columns inserted into the appropriate objects.

In Objectivity/DB, the code involving queries is much different. There is an Iterator class used by Objectivity/DB for Java, much like the one used for Java linked-list type classes (e.g., Vector), but that is where the similarity ends. Objectivity/DB can query on two different levels: database and container. A string predicate is built using regular expressions, which in some terms may be considered the SQL statement of the DOO-DBMS. The container or database is then sent the query and is told what type of objects to return. An Iterator is then used and the objects of interest are extracted. The difference is in the deliverables. In the relational case, the Resultset contains raw data that must be transformed into object form for Java to use. In the object-oriented database case, the deliverable is in fact an object that Java recognizes, along with all the relationships and inheritance hierarchy intact. As a result, method calls may begin on the data immediately and do not need to be preceded with "set" statements initializing all of the object's data.

5.2 Database Distribution

As seen above in the test results, database distribution is used in the testing of OOAIDE's performance. The appropriate autonomous partitions and databases images are copied to a second system, which is automatically updated when the primary image is updated. The distribution seems to have no ill effects on the overall system even when large amounts of objects are made persistent.

One point worth noting is the application's dependence on communications with the images. It would be very beneficial if the application could connect to just one image and the other image is updated by the DOODBMS itself. However, this is not the case. In order for any image to be updated, it has to participate in a quorum. This was one of the main reasons for distributing the weights between the images to where one is greater than the other, which establishes one image as the owner.

The communications burden on the network as a result of multiple images could be both small and large. In the case where the burden is small and network usage is optimal, objects are created and inserted into the database in very large numbers. Too much overhead is created when multiple objects cross the network in many packets. When many objects are sent at the same time, the number of packets decreases and network usage becomes more optimal. To measure the network load, traffic leaving the machine containing a tap would need to be monitored for packets destined for the database. Likewise, any traffic resulting from the TCP connection to the database back to the tap would also need to be monitored. Such a test is possible, but becomes difficult when the tap traffic must cross a heavily-loaded network such as the Internet. When multiple taps are connected to multiple databases over such a network, the problem of testing becomes even more difficult.

5.3 Demonstrate Taps Utilizing Common OOP Design

To demonstrate that a common design and language may be applied across a heterogeneous network on different types of IDSs, another tap is deployed on a Linux platform to retrieve data from the Snort IDS. Separate connections are made to the database from both the Real Secure and Snort taps and the database is distributed between two systems.

The flexibility of the design is the result of placing events from each tap in their own containers. Without this separation, write locks would need to be managed very carefully

so that multiple taps could access shared containers. However, the lock server is easily able to lock and manage two separate containers for two separate taps.

5.4 Summary

The chapter lays more credibility to the design of an IDS data repository using a DOODBMS. The results for insertion of data are very encouraging for high-speed transactions. Furthermore, the query results were better than expected given the simplicity of the queries and the fact that not many levels in the OOAIDE object hierarchy are traversed. The query results, although inconclusive in proving whether or not the design is the best for concurrency of multiple analysts workstations, does provide additional insight as to how the design could be modified to account for the types of queries that will most likely be used.

VI. Conclusions

In this thesis, an IDS data repository is successfully modeled, built, and tested. Initial tests indicate that the new OOAIDE meets, and in some cases, exceeds the performance levels of the current AIDE system.

6.1 Implementation Critique

There are a number of limitations discovered as a result of the OOAIDE development process. This section presents the limitations of the new object model and some advantages and disadvantages associated with OOAIDE.

6.1.1. AIDE to OOAIDE Translation Limitations

The implementation as given provides functionality in keeping with the functionality needed in an IDS data repository. It collects data, parses it, and sends it to the database in the form of objects. However, the model does not allow for a few key features that are used in AIDE. These features do not integrate well into the object model:

1. A number of tables in the AIDE model facilitate the counting of records in various forms. The ROLLED_UP_EVENT table discussed in the previous chapter is one example. In the object model, such objects may be used for this purpose, but it could probably be handled better at the container level. Objects to count objects are not considered the norm in object-oriented programming, but it may be the only way to solve the problem of getting certain pieces of key information quickly.

2. Checking objects against other objects upon insertion is very inefficient. The BOUNDARY table contains records from various firewalls and routers. When an EVENT is received, it is checked against the BOUNDARY table to see if it was actually blocked. In AIDE, this is done in the trigger as soon as the event is received for insertion. The OOAIDE tap does not perform this lookup due to its heavy network performance cost. To do so, the application would have to compare each and every event to all the objects of type Boundary over the network. This may be more efficient than at first glance due to the fact that entire pages are usually fetched from the database when a call is made and then cached by the application. However, there may be hundreds of thousands of objects to search and this type of correlation may better be suited to a process elsewhere on the network or on the DOODBMS site machine itself.

6.1.2. Accomplishments

The design does accomplish the key objectives listed in Chapter I:

1. *Performance.* OOAIDE shows an increase in performance in object insertions with the use of pure object data persistence and OO program integration. The overhead of building SQL statements and mapping all the attributes to specific columns is completely eliminated. Initial tests of traversing the object model to query for specific information also show promise. The preliminary retrieval results show that the two are close in their retrieval rates of specific objects, indicating that insertions of a large number of objects and distribution can exist without a notable decrease in performance in information retrieval.

2. *Distribution.* OOAIDE is successfully distributed between two machines on a network. The needed databases are successfully imaged and object insertions suffer no decrease in performance. Likewise, both Windows 2000 and Linux are used in the distributed tests demonstrating that distribution can be used on a heterogeneous network.
3. *Standardized Object-Oriented Model.* As a side-effect of using the DOO-DBMS, a standardized object model is built to accommodate the overall system to include both input and output of information. Using relationships, inheritance and normal methods, the application can be deployed in both a tap and analyst environment. And although Java is used as the sole language for the research presented in this thesis, C++ and Smalltalk are both supported by Objectivity/DB with no modifications to the OO database or its contents.

6.1.3. *Disadvantages*

With all of the functionality of the new model, there are still some disadvantages to consider. Although most of them do not pertain directly to the technical aspects, they should still be recognized as potential trouble spots should a system such as OOAIDE ever be deployed into an operational environment.

1. *Initial startup and DOODBMS familiarization.* Although a DOODBMS is a data store similar to an RDBMS, almost none of the terminology and interfaces are similar. Whereas an RDBMS is centered around tables and columns, the DOODBMS stores objects with attributes in containers and databases. Even the term database does not translate smoothly since it is considered to be

another type of collection object in a DOODBMS as opposed to the primary datastore encapsulating all aspects of the data as it is in an RDBMS. For this reason, anyone previously administering an RDBMS will need to become very familiar with object diagrams and the programming language to be used in building and administering the database.

2. *Maintenance.* From an object modeling perspective, data can be well encapsulated and relationships easily defined when designing the DOODBMS. However, using the model derived through the research in this thesis, maintenance of taps and other programs distributed throughout the federation could become cumbersome. If the DOODBMS changes, then the applications using the DOODBMS must also be changed to reflect the new object model.

6.2 Future Research

This thesis deals with the building of the architecture from the bottom up. This means that the database, although re-engineered from the original AIDE database, is designed around the data needed for a given tap to insert IDS information and retrieve information needed to establish relationships. There are many directions for future research to build upon the existing model, or develop an even better model. Below are listed just a few areas that should be considered:

- *Object retrieval.* This model does address the retrieval of objects from the database for use by the analyst, but a much more thorough job of testing needs to be accomplished before the operations are optimized. Such information will be imperative to the success of the overall model and storage decisions made in this

thesis. The decisions in this thesis are based mainly on concurrency involving the insertion of objects into the database to facilitate high-speed networks and modeling the system based solely on the existing AIDE system.

- *Server-side vs client-side processing of objects and correlation engine.* The foundation of this thesis involves the notion that distributing the load to the hosts performing the collection of data will allow the database to service more hosts over higher-speed networks. However, if multiple checks need to be made against objects within the database, then the load on the collecting machines may increase to a level that will impair their effectiveness as IDSs on the network. Therefore, correlation programs, processes that attempt to match events to boundaries or events to events, will probably need to be run on the server containing the database.

The correlations currently used by AIDE are highly simplistic in their approaches and are built around Oracle using the various correlation tables and ROLLED_UP_EVENT table. Further research is needed to assess what capabilities event correlators should contain and design characteristics for a system such as OOAIDE. At the very least, a correlation engine would need to be designed to correlate Events and Boundaries, as well as correlate Events to other Events. This can be done very efficiently on the database server given that a majority of the workload has been distributed throughout the system on other hosts. In theory, a much more advanced and robust application can be placed on the server versus the simple versions used in the current AIDE system.

- *Agents.* The taps built for OOAIDE are rather simple programs that are installed on a given machine and process data. What are really needed are autonomous agents that perceive their environment and react to it [Russell 7]. These agents could be sent out into the network with addresses of known IDSs, find those IDSs, install themselves on the host machines, and set up communications with a DOO-DBMS. The agents can also be programmed to monitor their respecting IDSs and report problems or other critical information. In the long run, they may even be used to administer certain portions of the IDSs to allow remote administrators the ability to manage multiple types of IDS sensors and repositories throughout the federation.

6.3 Conclusion

Object-oriented database technology is still a fairly young discipline despite advances over the past couple of decades. The relational model still continues to dominate the Air Force, and will probably continue to do so. However, in the area of high-speed networks where information needs to be placed in databases very quickly and efficiently, the distributed object model may prove better in terms of performance and pure distribution. The research presented in this thesis was not meant to fully establish OOAIDE as the only DOODBMS substitute for AIDE, but instead was meant to demonstrate that such a concept is not only feasible, but advantageous.

Appendix A. AIDE Data Dictionary

EVENT_CATEGORY : Stores unique corr_msg_id from EVENT_LOG.
CORR_MSG_ID VARCHAR2(40) NOT NULL : Correlated Msg ID mapped to EVENT_LOG
CORR_CATEGORY NUMBER(2) NOT NULL : Correlated Category mapped to
CORR_CATEGORY

SUPPORTED_SENSORS
SENSOR_NAME VARCHAR2(20) NOT NULL : Name of this sensor
SENSOR_TYPE VARCHAR2(1) NOT NULL : Type of sensor

SENSOR_VERSION
SENSOR_NAME VARCHAR2(20) NOT NULL : Mapped to SUPPORTED_SENSORS
VERSION VARCHAR2(20) NOT NULL : Version of this sensor

INCIDENT : Stores main detail for AIDE created incidents
INCIDENT_ID VARCHAR2(40) NOT NULL : Unique ID
USERLIST_ID NUMBER(14,0) : User that created incident. Mapped to USERLIST
INCIDENT_TITLE VARCHAR2(240) NOT NULL : Title for this incident.
ACCRYN VARCHAR2(20) NOT NULL : Mapped to SITE
CREATE_DT DATE DEFAULT sysdate : Date incident created
UPDATE_DT DATE DEFAULT sysdate : Date incident last updated
STATUS VARCHAR2(2) DEFAULT 'UI' : Status of incident
ASSET VARCHAR2(255) : Assets affected
IMPACT_SUM VARCHAR2(4000) : Summary of this incident's impact
NOTES VARCHAR2(4000) : Additional analyst notes
COUNTERMEASURES VARCHAR2(4000) : Countermeasures deployed due to incident
REF_INCIDENT_ID VARCHAR2(40) : Reference to another incident. Mapped to INCIDENT.

EVENT : Events reported by sensors/hosts.
EVENT_ID VARCHAR2(40) NOT NULL : Unique key identifying particular EVENT.
Combines incremented number and SITE acronym (ex. 1234_RL)
BOUNDARY_FOUND VARCHAR2(1) : Whether or not this EVENT was found in the
BOUNDARY table and thus indicated being intercepted
SITE_NAME VARCHAR2(20) NOT NULL : Mapped to SITE
CREATE_SNSR_DT DATE : Date sensor reported event
CREATE_DB_DT DATE : Date event inserted
UPDATE_DT DATE : Date event updated
SENSOR_NAME VARCHAR2(20) NOT NULL : Mapped to SENSOR. Sensor that inserted row.
SRCIP VARCHAR2(15) : Source IP of event
DESTIP VARCHAR2(15) : Destination IP of event
DESCRIPTION VARCHAR2(1024) : Description of event
SRCNAME VARCHAR2(125) : Resolution of source IP to network name
DESTNAME VARCHAR2(125) : Resolution of destination IP to network name
STAT VARCHAR2(2) : Status of event
SIGNATURE VARCHAR2(100) : Mapped to SENSOR_SIGNATURE
SIG_NAME VARCHAR2(40) : Mapped to AIDE_SIGNATURE
SRC_ID NUMBER(38,0) :
PRIORITY NUMBER(1,0) : Priority from AIDE_SIGNATURE

SITE_LOC VARCHAR2(20) NOT NULL : Mapped to SITE
SRCPORT NUMBER(38,0) : Source port of event
DESTPORT NUMBER(38,0) : Destination port of event
PROTOCOL VARCHAR2(10) : Protocol
PARTITION_VAL NUMBER(38,0) DEFAULT to_char(sysdate,'d') : Partition where event was placed within the database
TOT_COUNT NUMBER(10,0) : Total times this event has been reported

EVENT_LOG : Stores events that make up a correlated message
EVENT_LOG_ID NUMBER(12) NOT NULL : Unique event_log identifier
PARTITION_VAL NUMBER(2) NOT NULL : HR Inserted into DB
CORR_MSG_ID VARCHAR2(40) NOT NULL : Correlated Msg ID mapped to EVENT_CATEGORY
EVENT_ID VARCHAR2(40) NOT NULL : Event Id mapped to EVENT

EVENT_LOG_HIST : Long-term storage of past history of events that make up a correlated message
EVENT_LOG_ID NUMBER(12,0) NOT NULL
PARTITION_VAL NUMBER(2) NOT NULL : HR Inserted into DB
CORR_MSG_ID VARCHAR2(40) NOT NULL : Correlated Msg ID mapped to EVENT_CATEGORY
EVENT_ID VARCHAR2(40) NOT NULL : Event Id mapped to EVENT

EVENT_HIST : History table for events. Events are moved in here then deleted from the event table.
EVENT_ID VARCHAR2(40) NOT NULL : Unique key identifying particular EVENT. Combines incremented number and SITE acronym (ex. 1234_RL)
BOUNDARY_FOUND VARCHAR2(1) : Whether or not this EVENT was found in the BOUNDARY table and thus indicated being intercepted
SITE_NAME VARCHAR2(20) NOT NULL : Mapped to SITE
CREATE_SNSR_DT DATE : Date sensor reported event
CREATE_DB_DT DATE : Date event inserted
UPDATE_DT DATE : Date event updated
SENSOR_NAME VARCHAR2(20) NOT NULL : Mapped to SENSOR. Sensor that inserted row.
SRCIP VARCHAR2(15) : Source IP of event
DESTIP VARCHAR2(15) : Destination IP of event
DESCRIPTION VARCHAR2(1024) : Description of event
SRCNAME VARCHAR2(125) : Resolution of source IP to network name
DESTNAME VARCHAR2(125) : Resolution of destination IP to network name
STAT VARCHAR2(2) : Status of event
SIGNATURE VARCHAR2(100) : Mapped to SENSOR_SIGNATURE
SIG_NAME VARCHAR2(40) : Mapped to AIDE_SIGNATURE
SRC_ID NUMBER(38,0) :
PRIORITY NUMBER(1,0) : Priority from AIDE_SIGNATURE
SITE_LOC VARCHAR2(20) NOT NULL : Mapped to SITE
SRCPORT NUMBER(38,0) : Source port of event
DESTPORT NUMBER(38,0) : Destination port of event
PROTOCOL VARCHAR2(10) : Protocol
PARTITION_VAL NUMBER(38,0) DEFAULT to_char(sysdate,'d') : Partition where event was placed within the database
TOT_COUNT NUMBER(10,0) : Total times this event has been reported
FILENAME VARCHAR2(1024) :
EVENT_OS VARCHAR2(20)
USERNAME VARCHAR2(20)
LOGON_TYPE VARCHAR2(1)

USERLIST : This table is used to store data relative to info about identifiable system users authorized or not

USERLIST_ID NUMBER(14,0) NOT NULL : The username or login name associated with the known individual

USERNAME VARCHAR2(20) NOT NULL : The username affiliated with the password that Oracle tracks

FIRST_NAME VARCHAR2(20) : User's first name

MID_NAME VARCHAR2(10) : User's middle name or initial

LAST_NAME VARCHAR2(25) : User's last name

ADDRESS_1 VARCHAR2(40) : First work address line

ADDRESS_2 VARCHAR2(40) : Second work address line

CITY VARCHAR2(15) : City of work address

STATE VARCHAR2(3) : State of work address

ZIP VARCHAR2(10) : ip code of work address

COUNTRY VARCHAR2(10) : Country of work address

COMM_PHONE VARCHAR2(12) : User's work commercial phone number

DSN_PHONE VARCHAR2(12) : User's work DSN phone number

ALT_PHONE VARCHAR2(12) : User's alternate phone number

FAX_NUM VARCHAR2(12) : User's work related Facsimile phone number

ALT_FAX VARCHAR2(12) : User's alternate Facsimile phone number

PAGER_NOTIFY VARCHAR2(2) : Code identifying pager. Values are PH=Phone only, PC=Phone and Code, NP= No Pager

PAGER_NUM VARCHAR2(12) : Number dialed to access either the users pager or the pager service

PAGER_CODE VARCHAR2(15) : The additional code needed to address the users pager

EMAIL VARCHAR2(40) : User's work related e-mail address

ALT_EMAIL VARCHAR2(40) : User's alternate e-mail address

UNIT VARCHAR2(50) : User's unit

MAJCOM VARCHAR2(50) : Unit's MAJCOM

RANK VARCHAR2(50) : User's rank

STAT VARCHAR2(10) : The status of the user: 'active', 'inactive', 'unauth','unknown'

CREATE_DT DATE DEFAULT sysdate : The date the user's information was added to the system

TITLE VARCHAR2(30) : The person's official title, e.g. Analyst, Network Admin, etc.

HOST_SERVICE : Maintains relationships of hosts to its running services

HOST_SVC_ID VARCHAR2(40) NOT NULL

PORT_NUM NUMBER(10,0) : Port number of service

SERVICE_ID NUMBER : Mapped to service table

APPROVED VARCHAR2(1) : Whether or not the service is approved

HOST_ID NUMBER(14,4) NOT NULL : Mapped to HOST table

LICENSE_NUM VARCHAR2(100) : License number of service

DESCRIPTION VARCHAR2(200) : Description of service

PRODUCT_ID NUMBER(14,4) : Product ID of product running service

CREATE_DT DATE : Date entry was created

UPDATE_DT DATE : Date entry was updated

PROTOCOL VARCHAR2(5) : Protocol used by service

HOST : This table contains local domain registration info

HOST_ID NUMBER(14,4) NOT NULL : The primary key identifying a host record.
The decimal part of the key is the site number

HOSTNAME VARCHAR2(40) : The symbolic name given to identify the computer

IP_ADDR VARCHAR2(15) NOT NULL : The 15 character internet address in the format of nnn.nnn.nnn.nnn

MAC_ADDR VARCHAR2(20) : The hardware address on ethernet EPROM
in format xx:xx:xx:xx:xx:xx
HOST_IDENT VARCHAR2(40) : Unix only = results of the hostid command
OS_ID NUMBER(10,0) NOT NULL : The version of the primary OS running on the host
HARDWARE_TYPE VARCHAR2(15) : Description of hardware used (i.e. Sun, HP, Intel)
PORT_CHECK_DT DATE : Date of the last successful internal portscan (STROBE)
COORDINATE_ID NUMBER(12,2) : Pointer to the lat/long coordinates in the
COORDINATES table identifying the hosts position
ACRONYM VARCHAR2(20) : The acronym of the site where this host resides
STATE VARCHAR2(15) : State in which the host resides
POC_ID NUMBER(10,0) : Cross-reference to USERLIST for person of contact
SERIAL_NO VARCHAR2(50) : ADPE Serial Number
CREATE_DT DATE DEFAULT sysdate : Date entry was created
UPDATE_DT DATE DEFAULT sysdate : Date entry was last updated
FIXED_OS VARCHAR2(1) : If set to "Y", then CMU utility cannot over-write the OS_ID

CORR_CATEGORY: AIDE Correlation categories

CORR_CATEGORY NUMBER(2) NOT NULL : Correlated Msg Unique Identifier
PRIORITY_1_PERCENT NUMBER(6) : AIDE Green Priority percentage
PRIORITY_2_PERCENT NUMBER(6) : AIDE Yellow Priority percentage
PRIORITY_3_PERCENT NUMBER(6) : AIDE Red Priority percentage
CORR_TYPE VARCHAR2(80) : Category Type
DESCRIPTION VARCHAR2(240) : Description of Category

SERVICE: Contains services that may run on a host

SERVICE_ID NUMBER NOT NULL : Primary key identifying service
SERVICE_NAME VARCHAR2(25) NOT NULL : Name of service
PORT_NUM NUMBER NOT NULL : Port number service normally uses
PROTOCOL VARCHAR2(5) NOT NULL : Protocol service normally uses
SVC_DESC VARCHAR2(200) " Description of service
CREATE_DT DATE : Date entry was created
UPDATE_DT DATE : Date entry was updated

SITE: Contains information related to the Site(s) this AIDE instance serves

ACCRYN VARCHAR2(20) NOT NULL : Accronym of the site
NAME VARCHAR2(50) NOT NULL : Site Name
CERT VARCHAR2(20) : CERT Reporting to
DEFAULT_SITE VARCHAR2(1) :
SITE_IMG VARCHAR2(20) DEFAULT '/img/dash.gif' :
STATE VARCHAR2(2) : State
CITY VARCHAR2(20) : City
CREATE_DT DATE DEFAULT sysdate : Date created
UPDATE_DT DATE DEFAULT sysdate : Date updated
URL VARCHAR2(60) : URL of associated site
LAT NUMBER(10,2) : Latitude of Site
LON NUMBER(10,2) : Longitude of Site

ROLLED_UP_EVENT : This table stores unique srcip,destip,destport,sig_name from EVENT table.

It is updated from the EVENT_UPD trigger

ROLLUP_ID VARCHAR2(40) NOT NULL : This is the PK generated by a sequence
SRCIP VARCHAR2(15) : Source IP of the event
DESTIP VARCHAR2(15) : Destination IP of the event
DESTPORT NUMBER(38) : Destination Port of the event
UPDATE_DT DATE : Date of last occurrence of this group

CREATE_DT DATE : Date of first occurrence of group
SIG_NAME VARCHAR2(40) : AIDE Signature of event
TOT_COUNT NUMBER(10) DEFAULT 1 : Total number of occurrences of this group

SENSOR : Contains information about all possible Information Protection Sensors
SENSOR_NAME VARCHAR2(20) NOT NULL : Sensor Name mapped to SITE
and SENSOR_SIGNATURE
LOCATION VARCHAR2(20) NOT NULL : Where is the Sensor located
IP_ADDR VARCHAR2(15) : IP Address of the machine where the sensor is Located
ACCRYN VARCHAR2(20) NOT NULL : SITE abbreviation
STATUS VARCHAR2(1) DEFAULT 'F' NOT NULL : Status of the Bridge (T/F) Up or Down
PORT NUMBER(38) : Port where the Bridge runs
FULL_PATH VARCHAR2(80) : Full path to Sensor Bridge
ENCRYPTION VARCHAR2(3) : Is the Tap - Bridge encrypted?
CREATE_DT DATE : Date/Time the bridge was started
UPDATE_DT DATE : Heartbeat date/time sent to bridge

BOUNDARY : Information about all events captured by network sensors
EVENT_ID VARCHAR2(40) NOT NULL : Event ID (event_seq.nextval||"-"<||siteName)
SITE_NAME VARCHAR2(20) NOT NULL : Site name reporting event
SENSOR_NAME VARCHAR2(20) NOT NULL : The name sensor feeding the event information
LOCATION VARCHAR2(100) : Location of sensor
SIGNATURE VARCHAR2(100) : Sensor Signature
SIG_NAME VARCHAR2(40) : AIDE Signature
SITE_LOC VARCHAR2(20) : Site location reporting connection
PROTOCOL VARCHAR2(10) : Protocol of connection
SRCIP VARCHAR2(15) : Source IP
DESTIP VARCHAR2(15) : Destination IP
SRCNAME VARCHAR2(125) : Source Name
DESTNAME VARCHAR2(125) : Destination Name
PRIORITY NUMBER(38) : AIDE Priority
SRCPORT NUMBER(38) : Source Port
DESTPORT NUMBER(38) : Destination Port
CREATE_DB_DT DATE DEFAULT sysdate : Date/time inserted into DB
CREATE_SNSR_DT DATE : Date/Time sensor reported session
PARTITION_VAL VARCHAR2(2) DEFAULT to_char(sysdate,'d') : HR Inserted into DB

INCIDENT_EVENT

INCIDENT_ID VARCHAR2(40) NOT NULL : Mapped to INCIDENT
EVENT_ID VARCHAR2(40) NOT NULL : Mapped to EVENT

AIDE_SIGNATURE : AIDE Normalized signature data

SIG_NAME VARCHAR2(40) NOT NULL : The textual name for the unique signature
PRIORITY NUMBER(1,0) DEFAULT 1 NOT NULL : Ranks the level of concern associated
with the signature. Values are 1-3, where 1 is highest
CATEGORY VARCHAR2(40) : The grouping of the signature into a category
(e.g., Probe, Denial of service)
DESCRIPTION VARCHAR2(256) : The detailed explanation of the significance of the event
CREATE_DT DATE DEFAULT sysdate : Date the signature was created
UPDATE_DT DATE DEFAULT sysdate : Date the signature was last updated

SENSOR_TYPE

SENSOR_TYPE VARCHAR2(1) NOT NULL : Unique key identifying type of sensor
DESCRIPTION VARCHAR2(20) : Description of the sensor type

SENSOR_SIGNATURE

SENSOR_NAME VARCHAR2(20) NOT NULL : Mapped to SUPPORTED_SENSOR
SENSOR_SIG_NAME VARCHAR2(100) NOT NULL : The signature name that the sensor
is sending
SIG_NAME VARCHAR2(40) NOT NULL : AIDE Signature
TOT_COUNT NUMBER(38,0) : The running total of signature occurrences for each sensor
TOT_FALSE_ALARM NUMBER(38,0) : The total number of false alarms associated
with the reported signature
CREATE_DT DATE DEFAULT sysdate : The initial date the signature was recorded
UPDATE_DT DATE : The last time the signature event occurred

ARCHIVE : Detail of all archives performed via AIDE_Cleanup

ARC_DATE DATE NOT NULL
SITE_NAME VARCHAR2(20)
PARTITION VARCHAR2(2)
NUM_RECORDS NUMBER
EL_DELETES NUMBER(6)
EC_DELETES NUMBER(6)

SIGNATURE_CATEGORY : Stores correlation categories for AIDE_Signatures

SIG_NAME VARCHAR2(40) NOT NULL : Aide Signatures
CORR_CATEGORY NUMBER(38) NOT NULL : Correlated Category of the AIDE_Signature

Appendix B. OOAIDE Data Dictionary

Site

private ToManyRelationship hosts : One-to-many association to subordinate hosts (Host)
private ToOneRelationship parentSite : association to another site (Site)
private ToManyRelationship users : Many-to-many association to users (Userlist)

private String latitude : latitude of site
private String longitude : longitude of site
private long timeUpdate : last time of update
private long timeCreate : time of creation
private String acronym : acronym of site
private String name : name of site
private String cert : controlling CERT
private String siteImage : location of image
private String state : state where site is located
private String city : city where site is located
private String country : country where site is located
private String URL : URL of site

Host

private ToOneRelationship site : association to parent site (Site)
private ToManyRelationship sensors : One-to-many association to subordinate sensors (Sensor)
private ToManyRelationship hostServices : One-to-many association to hosts services (HostService)
private ToOneRelationship POC : One-to-one association to point of contact for this host (Userlist)
private boolean fixedOS : Whether or not this is a fixed OS
private short ip_A : first octet of IP address
private short ip_B : second octet of IP address
private short ip_C : third octet of IP address
private short ip_D : fourth octet of IP address
private String latitude : latitude of this host
private String longitude : longitude of this host
private long timePortCheck : last time the comm port was checked
private long timeUpdate : last time of update
private long timeCreate : time of creation
private String name : name of host
private String MACAddr : MAC address of host machine
private String osID : identifier for OS
private String ADPESerialNum : ADPE Serial Number
private String hardwareType : type of hardware host is running
private String hostIdent : identity of host via ident command

Sensor

private ToOneRelationship host : Many-to-one association to parent host (Host)

private ToOneRelationship supportedSensor : Many-to-one association to the type of sensor that this is (SupportedSensor)

private boolean encryption : whether or not this sensor encrypts its traffic
private short ip_A : first octet of IP address
private short ip_B : second octet of IP address
private short ip_C : third octet of IP address
private short ip_D : fourth octet of IP address
private int AMSPort : port that AMS runs on for distribution
private long timeUpdate : last time of update
private long timeCreate : time of creation
private char status : status of sensor (up or down)
private String name : name of sensor
private String location : location of sensor
private String fullPath : full pathname on sensor for files related to OOAIDE

HostService

private ToOneRelationship host : Many-to-one association to hosts (Host)
private ToOneRelationship service : Unidirectional association to service (Service)

private int portNum : port number associated with the service that the host is running
private Date timeUpdate : last time of update
private Date timeCreate : time of creation
private float productID : ID of product running service
private boolean approved : whether or not the service has been approved
private String licenseNum : license number
private String description : description of service
private String protocol : protocol service is running under (TCP, UDP, etc)

Service

private int portNum : port number under which this service normally runs
private long timeUpdate : last time of update
private long timeCreate : time of creation
private String name : common name of service
private String protocol : protocol under which this service normally runs
private String description : description of service

Activity

protected ToOneRelationship reportingSite : Unidirectional association to site (Site)
protected ToOneRelationship sensorSignature : Unidirectional association to signature that the sensor is reporting (SensorSignature)
protected ToOneRelationship sensor : Unidirectional association to sensor (Sensor)
protected ToOneRelationship aideSignature : Unidirectional association to cross-referenced AIDE signature (AIDESignature)

protected byte priority : priority from AIDE signature
private short ip_A : first octet of destination IP address
private short dstip_B : second octet of destination IP address
private short dstip_C : third octet of destination IP address
private short dstip_D : fourth octet of destination IP address
private short srcip_A : first octet of source IP address
private short srcip_B : second octet of source IP address
private short srcip_C : third octet of source IP address
private short srcip_D : fourth octet of source IP address

protected short dstPort : destination port of activity
protected short srcPort : source port of activity
protected long timeUpdate : last time of update
protected long timeCreate : time of creation
protected String srcName : resolution of source IP to network name
protected String dstName : resolution of destination IP to network name
protected String protocol : protocol of activity

Boundary : extends Activity

protected String info : additional information concerning boundary entry

Event

private ToOneRelationship category : Unidirectional association to category (Category)

private String status : status of event

private boolean boundaryFound : whether or not entry was previously identified as boundary object

Incident

private ToOneRelationship site : Unidirectional association to site (Site)

private ToManyRelationship events : Many-to-many Unidirectional association to events (Event)

private ToManyRelationship incidents : association to other incidents (Incident)

private long timeUpdate : last time of update

private long timeCreate : time of creation

private String ID : unique ID of incident

private String title : assigned title of incident

private String status : status of incident

private String asset : what assets were affected

private String impactSummary : summary of the impact incident had

private String notes : additional analyst notes

private String coutermeasures : countermeasures used

SupportedSensors

private ToManyRelationship sensorSignatures : Many-to-one association to the supported sensor's signatures (SensorSignature)

private String name : name of supported sensor

private String type : type of sensor

private String description : description for supported sensor

private String version : version of supported sensor

SensorSignature

private ToOneRelationship supportedSensor : Many-to-one association to supported sensors for signature (SupportedSensor)

private ToOneRelationship aideSignature : Unidirectional association to AIDE signature related to sensor signature (AIDESignature)

private long timeCreate : time of creation

private long timeUpdate : last time of update

private String name : name of signature

Category

private ToManyRelationship aideSignatures : Many-to-many association to AIDE signatures for category (AIDESignature)

private byte pri1 : OOAIDE green priority percentage
private byte pri3 : OOAIDE yellow priority percentage
private byte pri2 : OOAIDE red priority percentage
private String type : category type for events
private String description : description of category

AIDESignature

private ToManyRelationship categories : Many-to-many association to categories for this AIDE
signature (Category)

private byte priority :
private long timeCreate : time of creation
private long timeUpdate : last time of update
private String name : assigned name of signature
private String description : description of signature

Userlist

private ToOneRelationship site : Many-to-many association to the sites that a user has access to
(Site)

private int zip : zip code
private long timeCreate : time of creation
private long timeUpdate : last time of update
private String phoneComm : commercial phone number
private String phoneDsn : DSN phone number
private String phoneAlt : alternate phone number
private String phoneFax : fax number
private String phoneFaxAlt : alternate fax number
private String pagerNotify : code identifying pager
private String pagerNum : pager number
private String nameUser : user name
private String nameFirst : first name
private String nameMiddle : middle name
private String nameLast : last name
private String address1 : address line one
private String address2 : address line two
private String city : city of user
private String state : state of user
private String country : country of user
private String pagerCode : additional code needed to access user's pager
private String email : e-mail address
private String emailAlt : alternate e-mail address
private String unit : user's assigned unit
private String majcom : user's assigned MAJCOM
private String rank : rank of user
private String status : status (active, inactive, unauth, unk)
private String title : user's title (analyst, admin, etc)

Appendix C. Real Secure Tap/Trigger Code

This appendix contains code for both the AIDE tap, written in Perl, and the Oracle database trigger, written in PL/SQL, that inserts an event. It also contains, for comparison, the corresponding OOAIDE Java routine for executing the same function. Only lines of code that are significant to the task are included.

C.1 Perl/Oracle AIDE Real Secure Tap Code

Table 6-1. Perl Specifications for Real Secure Tap

Program/Package	Version	Description
Perl - ActiveState Distribution	5.6	Perl libraries used on 9.x/ME/NT/2000 platforms
Database Interface (DBI)	1.14	Perl database interface for use with relational databases (primarily with ODBC)
Oracle Database Interface (DBD::Oracle)	1.06	Perl database interface for use with Oracle 7 and 8 databases
ReadKey (Term::ReadKey)	2.14	A perl program for simple terminal control

```

while ($objDB->FetchRow()){
    @RS_Data = $objDB->Data;
    $ID = $RS_Data[0];
    $EventDate = $RS_Data[1];
    $SourceAddressName = $RS_Data[2];
    $DestinationAddressName = $RS_Data[3];
    $DestinationPortName = $RS_Data[4];
    $EventName = $RS_Data[5];
    $EngineIP = $RS_Data[6];
    $SourcePort = $RS_Data[7];
    $DestinationPort = $RS_Data[8];

    $Insert = $Oracle_DBI->prepare("INSERT INTO EVENT
(SITE_NAME,EVENT_ID,CREATE_SNSR_DT,SENSOR_NAME,SRICIP,
DESTIP,DESCRIPTION,
SIGNATURE,SITE_LOC,SRCPORT,DESTPORT)
VALUES ('$SITE_NAME',event_seq.nextval||'-'||'$SITE_NAME',
to_date('$EventDate','yyyy-mm-dd hh24:mi:ss'),
'$SENSOR_NAME','$SourceAddressName',
'$DestinationAddressName','$DestinationPortName',

```



```

        '$EventName', '$EngineIP', '$SourcePort',
        '$DestinationPort')")
    or die "Couldn't prepare statement: " . $Oracle_DBI->errstr;

    $status = $Insert->execute()
    or die "Couldn't execute statement: " . $Insert->errstr;

    &Commit_To_Aide;

////////////////////////////////////
////////////////////////////////////ORACLE TRIGGER////////////////////////////////////
////////////////////////////////////
    IF :new.sig_name IS NULL THEN
        IF :new.signature IS NOT NULL THEN
            -- Get sig_name from SENSOR_SIGNATURE --
            :new.sig_name
                :=sig_upd_function(:new.sensor_name, :new.signature);
            -- Go get priority AND category FROM AIDE_SIGNATURE --
            l_array := find_cat(:new.sig_name);
            -- Assign l_array(1) to PRIORITY --
            :new.priority := l_array(1);
        END IF;
    END IF;
    IF :new.sig_name IS NOT NULL THEN
        -- Update actual event IN EVENT table. --
        SELECT tot_count, update_dt INTO v_tot_count, v_update_dt
        FROM rolled_up_event
        WHERE srcip = NVL(:new.srcip, '0.0.0.0') AND
            destip = NVL(:new.destip, '0.0.0.0') AND
            destport = NVL(:new.destport, -1) AND
            sig_name = :new.sig_name;
        :new.tot_count := v_tot_count;
        :new.create_db_dt := v_update_dt;
    //////////////////////////////////////
    //////////////////////////////////////ORACLE FUNCTIONS////////////////////////////////////
    //////////////////////////////////////
    (P_SIG_NAME IN VARCHAR2)
    RETURN RETURN_TABLE_TYPE.ARRAYTYPE
    IS
BEGIN
    l_array(1) := 0;
    l_array(2) := 0;
    SELECT priority
    INTO l_array(1)
    FROM aide_signature
    WHERE sig_name = p_sig_name;
    RETURN l_array;
END FIND_CAT;

(P_SENSOR_NAME IN VARCHAR2 , P_SENSOR_SIG IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
    SELECT sig_name, tot_count
    INTO v_sig_name, v_tot_count
    FROM sensor_signature
    WHERE sensor_name = l_sensor_name
    AND sensor_sig_name = p_sensor_sig;
RETURN v_sig_name;
END sig_upd_function;

```

C.2 Java OOAIDE Real Secure Tap Code

```

while (true)
{
    try
    {

```

```

        Statement stmt = relCon.createStatement();
        results = stmt.executeQuery("SELECT ID, EventDate, DestinationAddressName,
DestinationPort, SourceAddressName, SourcePort, ProtocolID, EventName, EngineIP,
DestinationPortName, SourceAddress, DestinationAddress FROM RSLog WHERE ID > " +
index + ";");
    } catch (Exception e)
    {
        System.out.println("readData: SELECT: " + e);
    }
    try
    {
        session.begin();

        while (results.next())
        {
            index = results.getLong(1);
            java.util.Date eventDate = formatter.parse(results.getString(2),
pos);
            event = new Event(results.getString(3), "",
            new Long(results.getLong(4)).shortValue(),
            results.getString(5), "",
            new Long(results.getLong(6)).shortValue());
            sensorSig =
                (SensorSignature)sensorSignatureHashMap.get(results.getString(8));
            protocolID = results.getInt(7);
            if (protocolID == 0)
                event.setProtocol("TCP");
            else if (protocolID == 1)
                event.setProtocol("UDP");
            else
                event.setProtocol("UNK");
            eventCont.cluster(event);

            event.setReportingSite(site);
            event.setSensor(sensor);
            event.setTimeCreate(eventDate);

            if(sensorSig != null)
            {
                event.setSensorSignature(sensorSig);
                aideSig = sensorSig.getAIDESignature();
                if(aideSig != null)
                {
                    category = aideSig.getCategory();
                    event.setAIDESignature(aideSig);
                    event.setPriority(aideSig.getPriority());
                    if(category != null)
                        event.setCategory(category);
                }
            }
        }
    }
    results.close();
    session.commit();

    catch (InterruptedException i)
    {
        System.out.println("RSLog query exception (event data): " + i);
        i.printStackTrace();
        super.terminate();
    }

```

```
    }  
    catch (Exception e)  
    {  
        System.out.println("RSLog query exception (event data): " + e);  
        e.printStackTrace();  
        super.terminate();  
    }  
}
```

Appendix D. Query Test Code

D.1 All Events

D.1.1 AIDE

```
public void readRelDataOneJoin()
{
    int index = 0;

    Site site = new Site();
    Event event;
    java.util.Hashtable table = new java.util.Hashtable();

    ResultSet results = null;

    Statement stmt = null;

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());
    try
    {
        stmt = relCon.createStatement();

        results = stmt.executeQuery(
            "SELECT  S.NAME,  S.ACCRYN,  S.SITE_IMG,  S.STATE,  S.CITY,
S.CREATE_DT, S.UPDATE_DT, S.URL, S.LAT, S.LON, S.CERT, "
            + "E.CREATE_DB_DT, E.UPDATE_DT, E.SENSOR_NAME, E.SRCIP, E.DESTIP,
E.DESCRPTION, E.SRCNAME, E.DESTNAME, E.STAT, E.SIGNATURE, E.SIG_NAME, E.PRIOR-
ITY, E.SITE_LOC, E.PROTOCOL, E.SRCPORT, E.DESTPORT "
            + "FROM EVENT E, SITE S "
            + "WHERE (E.SITE_NAME = S.ACCRYN)"; // AND (S.ACCRYN = 'LZ')");

    }
    catch (Exception e)
    {
        System.out.println("readData: " + e);
    }

    try
    {
        while (results.next())
        {

            if (index == 0)
            {
                site.setName(results.getString(1));
                site.setAcronym(results.getString(2));
                site.setSiteImage(results.getString(3));
                site.setState(results.getString(4));
            }
        }
    }
}
```

```

        site.setCity(results.getString(5));
        site.setURL(results.getString(8));
        site.setLatitude(results.getString(9));
        site.setLongitude(results.getString(10));
        site.setCert(results.getString(11));
    }

    event = new Event();
    event.setSrcIP(results.getString(15));
    event.setDstIP(results.getString(16));
    event.setSrcName(results.getString(18));
    event.setDstName(results.getString(19));
    event.setStatus(results.getString(20));
    event.setPriority(results.getByte(23));
    event.setProtocol(results.getString(25));
    event.setSrcPort(results.getShort(26));
    event.setDstPort(results.getShort(27));

    event.setSite(site);

    table.put(site.getName(), event);

    index++;

    }
    results.close();
}

catch (Exception e)
{
    System.out.println("Query exception: " + e);
    e.printStackTrace();
    super.terminate();
}

java.util.Calendar time2 = java.util.Calendar.getInstance();
System.out.println(time2.toString());

System.out.println("" + index);
}

```

D.1.2 OOAIDE

```

public void readObjyDataOneLevel() {

    Site site = new Site();
    java.util.Hashtable table = new java.util.Hashtable();
    Event event;
    int index = 0;

    System.out.println("Looking for Events");
    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());

    session.begin();

    Iterator itr = eventCont.scan("Event");
}

```

```

while (itr.hasNext()){
    event = (Event)itr.next();

    if(site == null)
    {
        site = event.getReportingSite();
    }

    table.put(site.getName(), event);

    index++;
}

session.commit();

java.util.Calendar time2 = java.util.Calendar.getInstance();
System.out.println(time2.toString());
System.out.println(" " + index);
}

```

D.2 All Events with AIDE Signature

D.2.1 AIDE

```

public void readRelDataTwoJoin()
{
    int index = 0;

    Site site = new Site();
    Event event;
    AIDESignature aideSig;
    java.util.Hashtable table = new java.util.Hashtable();
    java.util.HashMap aideMap = new java.util.HashMap();

    ResultSet results = null;

    Statement stmt = null;

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());
    try
    {
        stmt = relCon.createStatement();

        results =
            stmt.executeQuery(
                "SELECT S.NAME, S.ACCRYN, S.SITE_IMG, S.STATE, S.CITY,
S.CREATE_DT, S.UPDATE_DT, S.URL, S.LAT, S.LON, S.CERT, "
                + "E.CREATE_DB_DT, E.UPDATE_DT, E.SENSOR_NAME, E.SRCIP,
E.DESTIP, E.DESCRPTION, E.SRCNAME, E.DESTNAME, E.STAT, E.SIGNATURE, E.SIG_NAME,
E.PRIORITY, E.SITE_LOC, E.PROTOCOL, E.SRCPORT, E.DESTPORT, "
                + "A.PRIORITY, A.CATEGORY, A.DESCRPTION, A.CREATE_DT,
A.UPDATE_DT, A.SIG_NAME "
                + "FROM EVENT E, SITE S, AIDE_SIGNATURE A "
                + "WHERE (E.SIG_NAME IS NOT null) AND "
            );
    }
}

```

```

        + "(E.SITE_NAME = S.ACCRYN) AND "
        + "(E.SIG_NAME = A.SIG_NAME)"; //(S.ACCRYN = 'LZ')");
    }
    catch (Exception e)
    {
        System.out.println("readData: " + e);
    }

    try
    {
        while (results.next())
        {
            aideSig = null;

            if (index == 0)
            {
                site = new Site();
                site.setName(results.getString(1));
                site.setAcronym(results.getString(2));
                site.setSiteImage(results.getString(3));
                site.setState(results.getString(4));
                site.setCity(results.getString(5));
                site.setURL(results.getString(8));
                site.setLatitude(results.getString(9));
                site.setLongitude(results.getString(10));
                site.setCert(results.getString(11));
            }

            event = new Event();
            event.setSrcIP(results.getString(15));
            event.setDstIP(results.getString(16));
            event.setSrcName(results.getString(18));
            event.setDstName(results.getString(19));
            event.setStatus(results.getString(20));
            event.setPriority(results.getBytes(23));
            event.setProtocol(results.getString(25));
            event.setSrcPort(results.getShort(26));
            event.setDstPort(results.getShort(27));

            if (!aideMap.containsKey(results.getString(33))
                && results.getString(33) != null)
            {
                aideSig = new AIDESignature();
                aideSig.setPriority(results.getBytes(28));
                aideSig.setDescription(results.getString(30));
                aideSig.setName(results.getString(33));

                aideMap.put(aideSig.getName(), aideSig);
            }

            event.setSite(site);
            event.setAIDESignatureRef(aideSig);

            table.put(site.getName(), event);

            index++;
        }
    }

```

```

        results.close();
    }

    catch (Exception e)
    {
        System.out.println("Query exception: " + e);
        e.printStackTrace();
        super.terminate();
    }
    System.out.println("" + index);
    java.util.Calendar time2 = java.util.Calendar.getInstance();
    System.out.println(time2.toString());
}

```

D.2.2 OOAIDE

```

public void readObjyDataTwoLevel() {

    Site site = new Site();
    java.util.Hashtable table = new java.util.Hashtable();

    Event event;
    AIDESignature aideSig = new AIDESignature();
    int index = 0;

    System.out.println("Looking for Events");

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());

    session.begin();

    java.util.HashMap aideMap = super.getAIDESignaturesHashMap();

    Iterator itr = eventCont.scan("Event");

    while (itr.hasNext()) {
        event = (Event)itr.next();

        if(site == null)
        {
            site = event.getReportingSite();
        }

        if(aideMap.containsValue(event.getAideSignature()))
        {
            index++;
            aideSig = (AIDESignature)event.getAideSignature();
            table.put(site.getName(), event);
        }
    }

    session.commit();

    java.util.Calendar time2 = java.util.Calendar.getInstance();
    System.out.println(time2.toString());
}

```



```

        System.out.println("" + index);
    }

```

D.3 Events of Single IP

D.3.1 AIDE

```

public void readRelDataIPOnly()
{
    int index = 0;

    Site site = new Site();
    Event event;
    java.util.Hashtable table = new java.util.Hashtable();

    ResultSet results = null;

    Statement stmt = null;

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());
    try
    {
        stmt = relCon.createStatement();

        results =
            stmt.executeQuery(
                "SELECT S.NAME, S.ACCRYN, S.SITE_IMG, S.STATE, S.CITY,
S.CREATE_DT, S.UPDATE_DT, S.URL, S.LAT, S.LON, S.CERT, "
                + "E.CREATE_DB_DT, E.UPDATE_DT, E.SENSOR_NAME, E.SRCIP,
E.DESTIP, E.DESCRPTION, E.SRCNAME, E.DESTNAME, E.STAT, E.SIGNATURE, E.SIG_NAME,
E.PRIORITY, E.SITE_LOC, E.PROTOCOL, E.SRCPORT, E.DESTPORT "
                + "FROM EVENT E, SITE S "
                + "WHERE (E.SRCIP = '192.168.0.2') AND "
                + "(E.SITE_NAME = S.ACCRYN)");
    }
    catch (Exception e)
    {
        System.out.println("readData: " + e);
    }

    try
    {
        while (results.next())
        {
            if (index == 0)
            {
                site.setName(results.getString(1));
                site.setAcronym(results.getString(2));
                site.setSiteImage(results.getString(3));
                site.setState(results.getString(4));
                site.setCity(results.getString(5));
                site.setURL(results.getString(8));
                site.setLatitude(results.getString(9));
                site.setLongitude(results.getString(10));
            }
        }
    }
}

```

```

        site.setCert(results.getString(11));
    }

    event = new Event();
    event.setSrcIP(results.getString(15));
    event.setDstIP(results.getString(16));
    event.setSrcName(results.getString(18));
    event.setDstName(results.getString(19));
    event.setStatus(results.getString(20));
    event.setPriority(results.getByte(23));
    event.setProtocol(results.getString(25));
    event.setSrcPort(results.getShort(26));
    event.setDstPort(results.getShort(27));

    event.setSite(site);

    table.put(site.getName(), event);

    index++;
}
results.close();
}

catch (Exception e)
{
    System.out.println("Query exception: " + e);
    e.printStackTrace();
    super.terminate();
}

java.util.Calendar time2 = java.util.Calendar.getInstance();
System.out.println(time2.toString());

System.out.println("" + index);
}

```

D.3.2 OOAIDE

```

public void readObjyDataIPOnly() {

    Site site = new Site();
    java.util.Hashtable table = new java.util.Hashtable();
    Event event;
    int index = 0;

    System.out.println("Looking for Events");

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());

    session.begin();

    String predicate = new String("srcip_A == 192 AND srcip_B == 168 AND srcip_C
== 0 AND srcip_D == 2");

    Iterator itr = eventCont.scan("Event", predicate);

```

```

while (itr.hasNext()){
    event = (Event)itr.next();

    if(site == null)
    {
        site = event.getReportingSite();
    }

    table.put(site.getName(), event);

    index++;
}

session.commit();

java.util.Calendar time2 = java.util.Calendar.getInstance();
System.out.println(time2.toString());
System.out.println("" + index);
}

```

D.4 Events of Single IP with Sensor

D.4.1 AIDE

```

public void readRelDataIPandSensor()
{
    int index = 0;

    Site site = new Site();
    Event event;
    Sensor sensor = new Sensor();

    java.util.Hashtable table = new java.util.Hashtable();

    ResultSet results = null;

    Statement stmt = null;

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());
    try
    {
        stmt = relCon.createStatement();

        results =
            stmt.executeQuery(
                "SELECTS.NAME,S.ACCRYN,S.SITE_IMG,S.STATE,S.CITY,S.CREATE_DT,
S.UPDATE_DT, S.URL, S.LAT, S.LON, S.CERT, "
                +
                "E.CREATE_DB_DT,E.UPDATE_DT,E.SENSOR_NAME,E.SRCIP,E.DESTIP,E.DESCRPTION,E.SRCNA
ME,E.DESTNAME,E.STAT,E.SIGNATURE,E.SIG_NAME,E.PRIORITY, E.SITE_LOC, E.PROTOCOL,
E.SRCPORT, E.DESTPORT, "
                + "R.SENSOR_NAME,R.LOCATION,R.IP_ADDR,R.ACCRYN,R.STATUS,
R.PORT,R.CREATE_DT,R.UPDATE_DT "
                + "FROM EVENT E, SITE S, SENSOR R "
                + "WHERE "
            );
    }
}

```

```

        + "E.SENSOR_NAME = R.SENSOR_NAME AND "
        + "E.SITE_NAME = S.ACCRYN AND "
        + "R.ACCRYN = S.ACCRYN AND "
        + "E.SRCIP = '192.168.0.2'");
    }
    catch (Exception e)
    {
        System.out.println("readData: " + e);
    }

    try
    {
        while (results.next())
        {
            if (index == 0)
            {
                site.setName(results.getString(1));
                site.setAcronym(results.getString(2));
                site.setSiteImage(results.getString(3));
                site.setState(results.getString(4));
                site.setCity(results.getString(5));
                site.setURL(results.getString(8));
                site.setLatitude(results.getString(9));
                site.setLongitude(results.getString(10));
                site.setCert(results.getString(11));

                sensor.setName(results.getString(28));
                sensor.setLocation(results.getString(29));
                sensor.setIP(results.getString(30));
                sensor.setStatus(results.getString(31).charAt(0));
            }

            event = new Event();
            event.setSrcIP(results.getString(15));
            event.setDstIP(results.getString(16));
            event.setSrcName(results.getString(18));
            event.setDstName(results.getString(19));
            event.setStatus(results.getString(20));
            event.setPriority(results.getBytes(23));
            event.setProtocol(results.getString(25));
            event.setSrcPort(results.getShort(26));
            event.setDstPort(results.getShort(27));

            event.setSite(site);
            event.setSensorRef(sensor);

            table.put(site.getName(), event);

            index++;
        }
        results.close();
    }

    catch (Exception e)
    {

```

```

        System.out.println("Query exception: " + e);
        e.printStackTrace();
        super.terminate();
    }

    java.util.Calendar time2 = java.util.Calendar.getInstance();
    System.out.println(time2.toString());

    System.out.println("" + index);
}

```

D.4.2 OOAIDE

```

public void readObjyDataIPandSensor() {

    Site site = new Site();
    java.util.Hashtable table = new java.util.Hashtable();

    Event event;
    Sensor sensor;

    int index = 0;

    System.out.println("Looking for Events");

    java.util.Calendar time1 = java.util.Calendar.getInstance();
    System.out.println(time1.toString());

    session.begin();

    String predicate = new String("srcip_A == 192 AND srcip_B == 168 AND srcip_C
== 0 AND srcip_D == 2");

    Iterator itr = eventCont.scan("Event", predicate);

    while (itr.hasNext()){
        event = (Event)itr.next();

        if(site == null)
        {
            site = event.getReportingSite();
        }

        sensor = event.getSensor();

        table.put(site.getName(), event);

        index++;
    }

    session.commit();

    java.util.Calendar time2 = java.util.Calendar.getInstance();
    System.out.println(time2.toString());
    System.out.println("" + index);
}

```

Bibliography

- AIDE. AIDE User's Manual Version 3.2. Draft ed. AFRL Rome Labs, NY: 2000.
- Anonymous. Maximum Linux Security. Indianapolis, IN: SAMS, 2000.
- Blaaha, Michael, and William Premerlani. Object-Oriented Modeling and Design for Database Applications. Upper Saddle River, NJ: Prentice Hall, 1998.
- Chang, D.T., and V. Srinivasan. "Object Persistence in Object-Oriented Applications." Systems Journal vol. 36, No. 1 (1997). Online. Internet. 11 Oct 2000. Available: <http://www.research.ibm.com/journal/sj/361/srinivasan.html>.
- Davie, Bruce S., and Larry L. Peterson. Computer Networks: A Systems Approach. San Fransisco, CA: Morgan Kaufmann Publishers, Inc., 1996.
- Domajnko, Tomaz, Matjaz B. Juric, Bostjan Brumen and Ivan Rozman. "How to Store Java Objects." Java Report. April 1999. Available: http://www.poet.com/news/in_the_press/java_report/041999.html.
- Farley, Jim. Java Distributed Computing. 1st ed. Morris Street, Sebastopol, CA: 1998.
- Hanushevsky, Andrew. "Mass Storage for BaBar at SLAC." Presentation. Stanford University, 8 Oct 1999.
- . "Objectivity Open File System." Presentation. Stanford University, 8 Oct 1999.
- Horstmann, Cay S. and Gary Cornell. Core Java: Volume II - Advanced Features. Upper Saddle River, NJ: Prentice Hall, 2000.
- SecureCom. "About SecureCom 6000." Online. Available: <http://www.intrusion.com/Products/sc6000.shtml>
- JIDS. Computer Software. Defense Informaion Systems Agency, 2001. Online. Available (DoD Only): <https://iase.disa.mil/tools/JIDS.html>
- Kroenke, David M. Database Processing: Fundamental, Design & Implementation. 7th ed. Upper Saddle River, NJ: Prentice Hall, 1999.
- Larson, James A. Database Directions. Upper Saddle River, NJ: Prentice Hall, 1995.

- Linhares Lima, Pedro A. "A Methodology for Reengineering Relational Databases to an Object-Oriented Database." Thesis. Air Force Institute of Technology. 1996.
- Loomis, Mary E.S. Object Databases: The Essentials. Reading, Mass.: Addison-Wesley, 1995
- Montgomery, Stephen. Building Object-Oriented Software. New York, NY: McGraw-Hill, 1998.
- Meyer, Steven. "Very Large Databases at Stanford Linear Accelerator Center." Technical Report. Stanford University, 28 May 1997.
- Muller, Pierre-Alain. Instant UML. Paris, France: WROX Press, 1997.
- NetRadar. Computer Software. Net Squared, Inc., 2001. Online. Available: <http://www.netsq.com/Tools/NetworkRadar/index.php3>
- NFR. Computer Software. Network Flight Recorder Security, Inc., 2001. Online. Available: <http://www.nfr.com>
- Nmap. Computer Software. Insecure.org, 2001. Online. Available: <http://www.insecure.org/nmap>
- Nessus. Computer Software. 2001. Online. Available: <http://www.nessus.org>
- Objectivity. Objectivity/DB Technical Overview. Objectivity Inc., 1999.
- . Objectivity/C++. Using Objectivity/C++: Version 4. Objectivity Inc., 1996.
- . Objectivity/Admin. Objectivity/DB Administration. Objectivity Inc., 1996.
- . Objectivity/FTO-DRO. Objectivity/FTO and Objectivity/DRO. Objectivity Inc., 1999.
- . Objectivity/Java. Objectivity for Java Guide Release 6.0. Objectivity Inc., 2000.
- Oracle Corporation. Oracle8i Enterprise Edition: Getting Started Release 8.1.5 for Windows NT. Redwood Shores, CA: Oracle Corporation, 1999.
- Ranum, Marcus J. "Intrusion Detection & Network Forensics." Presentation. 12th Annual FIRST Conference (2000), June 1999.
- Raptor. Computer Software. Symantec, Corp., 2001. Online. Available: <http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=47>
- RealSecure. Computer Software. Internet Security Systems, Inc., 2001. Online. Available: <http://www.iss.net>
- Sidewinder. Computer Software. Secure Computing Corporation, 2001. Online. Available: <http://www.securecomputing.com>

Veridian - Trident Data Systems. CIDDS Installation, Administration and User's Manual
Version 3.1. Draft ed. San Antonio, TX: Veridian, 2000.

Wu, Hsin-feng (Edward). "A Distributed Object-Oriented Database Application Design."
Air Force Institute of Technology. 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 03-07-2001		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From - To) Jun 2000 - Mar 2001	
4. TITLE AND SUBTITLE USING A DISTRIBUTED OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM IN SUPPORT OF A HIGH-SPEED NETWORK INTRUSION DETECTION SYSTEM DATA REPOSITORY				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
				5d. PROJECT NUMBER 00-034	
6. AUTHOR(S) Polk, Phillip W., 2Lt, USAF				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-09	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Bldg 640 WPAFB, OH 45433-7765				10. SPONSOR/MONITOR'S ACRONYM(S)	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFGB Attn: Mr. John Feldman Information Directorate/Defensive Information Warfare Branch Air Force Research Laboratory 525 Brooks Rd. Rome, NY 13441-4505 COMM:(315) 330-2664				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The Air Force has multiple initiatives to develop data repositories for high-speed network intrusion detection systems (IDS). All of the developed systems utilize a relational database management system (RDBMS) as the primary data storage mechanism. The purpose of this thesis is to replace the RDBMS in one such system developed by AFRL, the Automated Intrusion Detection Environment (AIDE), with a distributed object-oriented database management system (DOODBMS) and observe a number of areas: its performance against the RDBMS in terms of IDS event insertion and retrieval, the distributed aspects of the new system, and the resulting object-oriented architecture. The resulting system, the Object-Oriented Automated Intrusion Detection Environment (OOAIDE), is designed, built, and tested using the DOODBMS Objectivity/DB. Initial tests indicate that the new system is remarkably faster than the original system in terms of event insertion. Object retrievals are also faster when more than one association is used in the query. The database is then replicated and distributed across a simple heterogeneous network with preliminary tests indicating no loss of performance. A standardized object model is also presented that can accommodate any IDS data repository built around a DOODBMS architecture.					
15. SUBJECT TERMS Intrusion Detection System, Object-Oriented, Databases, Object-Oriented Database, Distributed Database, Distributed Object-Oriented Database, Data Repository, Object-Oriented Programming					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 160	19a. NAME OF RESPONSIBLE PERSON Dr. Gregg H. Gunsch, ENG
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-6565, ext 4281